TURING 图灵程序设计丛书

MANNING

MongoDB in Action

MongoDB 实战

[美] Kyle Banker 著
丁雪丰 译

- MongoDB开发者现身说法
- 由浅入深、注重实践
- 涵盖MongoDB开发及运维





人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



Kyle Banker

软件工程师，曾工作于10gen公司，负责维护Ruby及C语言的官方MongoDB驱动、领导MongoDB文档项目并开发培训课程，且为客户提供咨询、商业支持和培训；现任职于Snapjoy（为用户提供默认私有的在线照片备份和自动管理服务）。个人网站：http://kylebanker.com/blog。



丁雪丰

一线“攻城师”一枚，InfoQ中文站小编，满江红翻译组核心成员，常年混迹于各种社区，业余时间写作、翻译、汉化软件，《RESTful Web Services Cookbook中文版》、《Spring攻略》等多部书的译者。

图灵程序设计丛书

MongoDB in Action

MongoDB实战



[美] Kyle Banker 著
丁雪丰 译

人民邮电出版社
北京

图书在版编目（CIP）数据

MongoDB实战 /（美）班克（Banker,K.）著；丁雪丰译. -- 北京：人民邮电出版社，2012.10
（图灵程序设计丛书）
书名原文：MongoDB in Action
ISBN 978-7-115-29507-1

Ⅰ. ①M… Ⅱ. ①班… ②丁… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2012)第232104号

内 容 提 要

本书由10gen公司（开发并支持开源数据库MongoDB）负责维护Ruby及C语言官方MongoDB驱动的软件工程师Kyle Banker编写而成，是一本全面细致介绍MongoDB及其应用的权威指南。本书共分三部分，首先介绍MongoDB的历史、特性和使用场景，然后细致阐述MongoDB API，专注于应用程序开发渐近式描述电子商务应用的模式与操作，并最后从DBA的角度考量性能和运维。另外，书中还介绍了面向文档数据库模型，并深度剖析了复制、自动分片以及部署等特性。

本书适合初中级应用程序开发者和DBA学习参考。

图灵程序设计丛书
MongoDB实战

◆ 著　　[美] Kyle Banker
　译　　丁雪丰
　责任编辑　毛倩倩
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷
◆ 开本：800×1000　1/16
　印张：16.25
　字数：394千字　　2012年10月第1版
　印数：1－3 500册　　2012年10月北京第1次印刷
著作权合同登记号　图字：01-2012-4879号
ISBN 978-7-115-29507-1

定价：59.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

谨以此书献给那些为和平和人性尊严而奋斗的人们。

译　者　序

骐骥一跃，不能十步；驽马十驾，功在不舍。

——《荀子·劝学》

IT是个知识更新十分迅速的行业，IT人士除了掌握基础知识，还要经常关心技术动态和身边不断涌现出的新技术，不然就会有落伍的可能。作为一个典型的摩羯男，我有着一颗不安于现状的心，想要不断超越自己。要收获就得有付出，还是努力充电吧，打好基础才能有资本去迎接挑战。比如，Pragmatic Programmers建议每年学习一门新语言①，所谓他山之石可以攻玉，就算没机会在工作中用到它，其解决问题的思路也值得借鉴。

我的选择有所不同，大约五年前，我给自己定下了一个目标，在自己30岁之前，每年翻译一本书。一来借翻译之机深入学习一些东西，二来可以帮助更多的同行。也许是摩羯的坚持，经过几年的努力，我终于可以为这个计划画上一个句号了。没错，你所看到的这本书就是我30岁计划的收官之作。希望本书能帮助你了解、学习、掌握MongoDB，如果能帮助在工作中解决实际的问题，那就再好不过了。

"云计算"、"大数据"和"NoSQL"都是近年的热点名词，本书的"主人公"MongoDB和这些名词都能扯上关系，加之它和传统的关系型数据库有着这么多相似之处，实在是没办法忽略它的存在，就算用不上，也该好好了解一下。在网上读了不少文章，也对MongoDB有了一个大概的认识之后，是不是会期待有一本书能将众多知识点集于一身，层层深入，融会贯通？《MongoDB权威指南》当然不容错过，不过薄薄一本小册子难免无法深入展开，而且从它出版之后，MongoDB也发生了不少重大变化。也许可以考虑一下这本书，本书作者同样来自MongoDB背后的公司10gen，而且写作上符合in Action系列的一贯风格，内容由浅入深，注重实践。无论你是想了解MongoDB的使用方法还是具体实现细节，无论你是开发者还是DBA，都能在书中找到需要的内容。

我总是喜欢看些"失败案例"或者"重大故障"，比如著名的Foursquare在MongoDB上就吃过不少苦头，具体的细节我就不再赘述了。总之，在部署MongoDB时一定要多加小心。那么到底该如何进行调优，怎么处理复制和分片，怎么维护集群呢？如果对MongoDB已经有所了解，

---

① 以前一直以为这是Martin Fowler的建议，在自己写的第一篇译者序里还用到了这句话，但在上次Martin Fowler来华时，他指出这其实是Pragmatic Programmers提出的。——译者注

可以直接翻到对应的章节，本书的后半部分每章都独立成篇，你可以直奔主题，选择性地阅读需要的部分。

在翻译过程中，不断有人问我什么时候可以出版，我只能回答还得再等等，因为我从年初开始花了整整7个月的时间才完成了全书的翻译，所以只能对读者说抱歉了。本书能与各位读者见面，离不开图灵公司各位老师的辛勤付出，还有我的各位好友在翻译过程中提供的各种建议。今天正好是七夕，在这个特殊的日子里，我还要感谢一下我的新婚妻子，她一直默默支持我，让我做自己想做的事，本书也有她的一份功劳。

虽然花了这么多时间仔仔细细地进行翻译，不过碍于本人水平有限，如果你在阅读过程中发现什么问题，还望能够不吝赐教，比如通过图灵社区（ituring.com.cn），或者新浪微博（@DigitalSonic），先行谢过各位。

丁雪丰

2012年8月23日于上海

前　　言

数据库是信息时代的“老黄牛”，就像希腊神话中的擎天神Atlas[①]一样，它们默默地支撑着我们赖以生存的数字世界。发布评论和微博，乃至查找并排序内容，这些操作从本质上来说都是和数据库打交道，而我们恰恰会对此熟视无睹。正因为这个基础的“隐蔽功能”，我总是会对数据库心存敬畏，这种敬畏和走过本来只让汽车通行的悬索大桥时所产生的敬畏没什么分别。

数据库有很多种形式。图书馆里的图书目录和卡片分类都算是其中的一种，昔日Perl程序员使用的特殊结构的文本文件也是。也许现在最广为人知的数据库，就是功能丰富、让人赚得盆满钵盈的关系型数据库了，它支撑着这个世界上的很多软件。这些关系型数据库，连同它们那理想化的第三范式和富于表达力的SQL接口，仍然让那些保守派肃然起敬。

但是，作为一名有几年工作经验的Web应用程序开发者，我渴望尝试一些能替代占据统治地位的关系型数据库的后起之秀，发现MongoDB之后，便对它爱不释手。MongoDB使用类似JSON的结构来表示数据，我喜欢这个设计。JSON简单、直观而且易用。MongoDB还将其查询语言构建于JSON之上，使得这个新数据库在使用上很舒适很协调。接口之外的一些引人注目的特性让它更具魅力，例如方便复制和分片。我使用MongoDB构建了一些应用程序，亲身体验了它带给开发的舒适性之后，便深深爱上了MongoDB。

机缘巧合，我加入了10gen——领导开发开源数据库MongoDB的公司。两年来，我有机会改善多款客户端驱动，与众多客户一起部署他们的MongoDB。我希望在这一过程中所积累的经验都能原汁原味地体现在你正阅读的这本书中。

作为一款还在不断完善的作品，MongoDB还有很长的路要走，但它已经成功地支撑了成百上千的应用程序，运行在大大小小的数据库集群之上，而且每天都在进步。MongoDB每天都能为不少开发者带来惊喜，甚至是幸福，希望你也能拥抱MongoDB，感受它的魅力。

---

① Atlas是希腊神话中的擎天神，因背叛宙斯被降罪用双肩支撑苍天。——译者注

致　　谢

我要感谢Manning的几位人士帮助我将本书变为现实，感谢Michael Stephens帮我构思，感谢策划编辑Sara Onstine和Jeff Bleiel的一路支持。感谢他们。

写书是件很耗时的事，要没有Eliot Horowitz和Dwight Merriman的宽宏大量，我很可能都没有时间来完成本书。Eliot和Dwight别出心裁地创造了MongoDB，而且放心地让我来为这一项目编写文档。感谢他们。

书中的很多主意都来源于我和10gen的同事们的交谈。为此，要特别感谢Mike Dirolf、Scott Hernandez、Alvin Richards和Mathias Stearn。另外，我还要感谢Kristina Chodorow、Richard Kreuter和Aaron Staple为全书所有章节提供的专业建议。

下列审稿人在本书的不同编写阶段中阅读了手稿，我要感谢他们提供了颇有价值的反馈：Kevin Jackson、Hardy Ferentschik、David Sinclair、Chris Chandler、John Nunemaker、Robert Hanson、Alberto Lerner、Rick Wagner、Ryan Cox、Andy Brudtkuhl、Daniel Bretoi、Greg Donald、Sean Reilly、Curtis Miller、Sanchet Dighe、Philip Hallstrom和Andy Dingley。还要感谢Alvin Richards在付梓前夕对本书终稿做的全面技术审校。

我为我的妻子Dominika和儿子Oliver感到骄傲。谢谢Dominika的耐心和支持，谢谢Oliver，他真是棒极了。

关 于 本 书

本书适合那些想从基础开始了解MongoDB的应用程序开发者和DBA学习参考。如果你刚刚接触MongoDB，会发现本书是很好的教材，内容由浅入深。如果你已经是一位MongoDB用户了，本书的详细参考指南部分一定能助你一臂之力，它能弥补你知识点上的空白。从深度上来说，本书内容适合资深高级用户之外的所有用户。

本书的代码示例使用的是JavaScript和Ruby，前者是MongoDB Shell的语言，后者是流行的脚本语言。书中尽可能提供简单、有用的示例，只使用JavaScript和Ruby中最普通的特性，主要目的是以最易理解的方式展现MongoDB API。如果你用过其他编程语言，会发现这些例子都很容易理解。

关于语言，还有一点需要说明。如果你心存疑惑："为什么本书不使用某某语言？"那么大可不必担心。官方支持的MongoDB驱动提供了一致且类似的API，这意味着一旦你了解了某款驱动的基本API，很快就能上手其他的驱动。方便起见，本书在附录D中提供了对PHP、Java和C++驱动的概述。

本书内容

本书既是教程，又是参考指南。如果你刚刚接触MongoDB，按顺序阅读全书定会大有收获。书中有大量代码示例，你可以自行运行它们以巩固对概念的理解。运行这些示例前，你至少需要安装MongoDB，最好还有Ruby驱动，附录A中有相关的安装指南。

如果你已经用过MongoDB，那么可能会对某些特定的主题更感兴趣。第7章到第10章，以及所有的附录都独立成篇，可以跳跃阅读。此外，第4章到第6章关注于基础知识，它们也能脱离上下文进行阅读。

本书结构

本书分为三部分。

第一部分是对MongoDB的一个详细介绍。第1章概述了MongoDB的历史、特性及使用场景。第2章通过MongoDB命令界面介绍了这一数据库的核心概念。第3章介绍了一个在后端使用MongoDB的简单应用程序的设计。

第二部分对第一部分中用到的MongoDB API做了详细说明。这部分共三章，特别专注于应用程序开发，渐进式地描述了电子商务应用的Schema和操作。第4章专门讲解MongoDB中最小的数据单元——文档，提供了一套基本的电子商务Schema。第5章和第6章讲述如何通过查询和更新来使用该Schema。为了更好地进行说明，第二部分中的每一章都对相应主题作了条分缕析的讲解。

第三部分关注性能和运维。第7章彻底研究了索引和查询优化。第8章聚焦于复制，讨论高可用性和读可扩展的MongoDB部署策略。第9章介绍MongoDB的水平扩展方法——分片。第10章是一系列最佳实践，包含部署、管理以及MongoDB安装的疑难解答。

本书最后还有5个附录。附录A涉及了MongoDB和Ruby（用于演示驱动）在Linux、Mac OS X和Windows上的安装。附录B介绍了一系列Schema和应用程序设计模式，还包含了一组反模式。附录C演示了如何在MongoDB中使用二进制数据，以及如何使用GridFS（所有驱动都实现了的一个规范）在数据库中存储大文件。附录D对PHP、Java和C++的驱动做了一个比较研究。附录E演示了如何使用空间索引（spatial indexing）来查询地理坐标。

代码约定与下载

书中出现的所有源代码都用等宽字体表示，借此区别于普通文字。

有些代码清单带有代码注解以突出重要概念，有些地方还有带数字编号的项目符号，以与下文的解释相联系。

作为一个开源项目，10gen将MongoDB的问题追踪系统开放给了社区。书中的很多地方，尤其是脚注里，常有问题报告和计划改进的引用。举个例子，为数据库添加全文搜索的问题单是SERVER-380。要查看该问题单的状态，可以通过浏览器访问http://jira.mongodb.org，在搜索框中输入单号。

你可以从本书的网站http://mongodb-book.com以及原出版社的网站http://manning.com/MongoDBinAction下载本书的源代码[①]和示例数据。

软件要求

想要最大限度地利用本书，你需要在自己的系统上安装MongoDB，可以在附录A和MongoDB官方网站（http://mongodb.org）中找到安装指南。

如果想要运行Ruby驱动的例子，那么还需要安装Ruby，同样可以参考附录A中的安装指南。

作者在线

本书的读者还可访问Manning Publications运营的私有论坛，在论坛中评论本书、询问技术问

① 也可在图灵社区（www.ituring.com.cn）本书网页免费注册下载。——编者注

题以及寻求作者和其他用户的帮助。要访问并订阅该论坛，请在浏览器中访问www.manning.com/MongoDBinAction并单击Author Online，这个页面中提供的信息包括注册后如何访问论坛、可以获得哪些帮助，还有论坛的管理规则。

Manning承诺为读者之间和读者与作者之间的交流提供场所，但对作者在论坛中的参与程度并不做要求，他是义务（不计报酬）参与本书论坛的。我们建议你尝试问他一些有挑战性的问题，让他有兴趣继续访问本论坛。

只要本书英文版在销售，作者在线论坛的内容以及之前讨论的存档都会保留在出版社的网站上。

关于封面图片

本书封面图片名为“Le Bourginion”，即法国东北部勃艮第地区的居民，取自法国出版的四卷地方服饰风俗概要十九世纪版，作者是Sylvain Maréchal。其中，每张图都画得很精致并手工上色。Maréchal作品中收集的服饰种类众多，生动地呈现了200年前世界上地区和城镇在文化上的差异。地区之间相互隔离，人们说着不同的方言。在城市或者乡下，很容易就能通过衣服分辨出这人生活在哪里，他是做什么的以及他的身份地位。

自那以后，服饰的风俗发生了变化，当时丰富的地区多样性也已消失殆尽。现在很难区分出不同大洲、地区或城镇的居民了。也许我们是用文化多样性换取了更多样的人生——无疑是更多样、更快节奏的科技生活。

Maréchal呈现给大家的图片体现了两个世纪前地区生活丰富的多样性，在计算机图书很难相互区别的今天，Manning的图书借封面来彰显其计算机图书的独创性和主动性。

目　　录

Part 1

第一部分

入门指南

这部分是一个宽泛、实用的 MongoDB 入门教程，此外还介绍了 JavaScript Shell 和 Ruby 驱动，全书的示例都会用到它们。

第 1 章将回顾 MongoDB 的历史、设计目标以及应用场景。我们还会拿它和其他“NoSQL”领域的新兴数据库做对比，了解一下它的独一无二之处。

第 2 章里你将熟悉 MongoDB Shell 的语言，了解基本的 MongoDB 查询语句，并通过创建、查询、更新和删除文档来进行实践。这一章还会介绍一些高级 Shell 技巧和 MongoDB 命令。

第 3 章将介绍 MongoDB 驱动和数据格式——BSON。本章里你将了解到如何通过 Ruby 编程语言与数据库进行交互，并用 Ruby 构建一个简单的应用程序，该示例演示了 MongoDB 的灵活性及其强大的查询功能。

第1章 为现代Web而生的数据库

本章内容

- MongoDB的历史、设计目标和关键特性
- MongoDB Shell和驱动的简要介绍
- MongoDB的使用场景及其局限性

近几年，如果构建Web应用程序，你可能会选择关系型数据库作为主要数据存储方案，而且它的表现通常也能接受。大多数开发者都熟悉SQL，能体会到精心正规化（normalized）后的数据模型所散发出的美感，了解事务的必要性，知道持久化存储引擎提供的保证。就算我们不喜欢直接和关系型数据库打交道，也能找出很多工具帮助我们降低复杂度，上至管理控制台，下至对象关系映射器。简言之，关系型数据库十分成熟且有口皆碑。因此，当一小群有主见的骨干开发者开始提倡另一种数据存储时，便有人提出了关于这些新技术的可行性和实用性的问题。这些新数据存储是关系型数据库的替代品吗？谁在生产环境中使用它们，为什么选择它们？在向非关系型数据库的迁移过程中又要做哪些权衡？上述问题的答案都可以建立在这个问题的答案上：为什么开发者对MongoDB感兴趣？

MongoDB是一款为Web应用程序和互联网基础设施设计的数据库管理系统。MongoDB的数据模型和持久化策略的设计目标是提供高读写吞吐量，在易于伸缩的同时还能进行自动故障转移。无论应用程序只需要一个还是几十个数据库节点，MongoDB都能提供惊人的性能。如果你对扩展关系型数据库的艰辛深有体会，一定会觉得这是个好消息。但并非每个人都需要扩展数据库，也许需要的就只是一台数据库服务器，那么为什么要使用MongoDB呢？

MongoDB之所以一下子这么引人注意，并不是因为它的扩展策略，而是因为它那直观的数据模型。假设基于文档的数据模型可以表示丰富的、有层级的数据结构，那么抛弃关系型数据库所强加的复杂的多表关联就成为了可能。举例来说，假设你正在为一个电子商务网站做产品建模，如果使用完全正规化的关系型数据模型，任何产品的信息可能都会被打散到多张表中。如果想要从数据库Shell里获得产品表述，我们需要写一句由join堆砌而成的复杂SQL查询。其结果就是，大多数开发者需要依赖软件中的辅助模块将数据组装成有意义的东西。

相比之下，使用文档模型的话，大多数产品信息都能放在一个文档里。打开MongoDB

JavaScript Shell，可以轻松获得产品的完整表述，所有信息都按层级用一种类似JSON[1]的结构组织在一起。对于这样组织的所有信息，既可以做查询，也可以做其他操作。MongoDB的查询是专门为操作结构化文档而设计的，因此从关系型数据库切换过来的用户能有与之前类似的查询体验。此外，大多数开发者现在都使用面向对象的语言，他们想要一个能更好地映射到对象的数据存储。有了MongoDB，编程语言中定义的对象能被“原封不动”地持久化，消除掉一些对象映射程序的复杂性。

如果你对表列数据（tabular）和数据的对象表示之间的区别还很陌生，那么肯定会有很多问题。在本章末尾，我将给出MongoDB的特性和设计目标的完整概述，让你更清楚地明白为什么像Geek.net（SourceForge.net）和纽约时报（The New York Times）这样的公司的开发者要在他们的项目中使用MongoDB。我们将了解MongoDB的历史，认识它的主要特性。接下来，我们还要了解一些其他的数据库解决方案和所谓的NoSQL运动[2]，我会解释MongoDB在其中发挥的作用。最后，我还将概括说明MongoDB适用于哪些场景，在哪些场景下其他数据存储又可能会更合适一些。

1.1 生于云端

MongoDB的历史虽然不长，但却值得回顾，它诞生于一个更宏伟的项目。在2007年年中，一个名为10gen的创业公司着手开发一个PaaS（Platform-as-a-Service）项目，它由应用服务器和数据库组成，用于托管Web应用程序并能按需伸缩。与谷歌的AppEngine类似，10gen的平台也设计成能够自动伸缩，自动管理硬件和软件基础设施，它解放了开发者，让他们能够专注于应用程序代码。10gen最终发现大多数开发者并不喜欢放弃对技术栈的掌控，但他们的确喜欢10gen的新数据库技术。后来10gen将精力集中到数据库上，就有了MongoDB。

随着越来越多的人在大大小小的项目中选择MongoDB并在生产环境中进行部署，10gen继续以开源项目的形式支持MongoDB数据库的开发。代码是公开的，而且可以自由修改和使用，只要遵循其开源协议的条款即可，而且10gen也鼓励社区报告缺陷和提交补丁。到目前为止，MongoDB的所有核心开发者不是10gen的创始人，就是10gen的员工，而这一项目的规划继续由用户社区的需求来决定，创造该数据库的最终目标是将关系型数据库中最好的特性和分布式键值存储结合起来。因此10gen的商业模式和其他知名开源公司毫无二致：支持开源产品的开发，并向最终用户提供订阅服务。

这段历史中有几点需要注意。首先，MongoDB最初是为一个要求数据库能在多台机器间优雅伸缩的平台而开发的。其次，MongoDB是作为Web应用程序的数据存储设计的。正如我们稍后会看到的，MongoDB被设计为可水平伸缩的主要数据存储，这一点把它和其他现代数据库系统区别开来。

---

① JSON是JavaScript Object Notation的缩写。我们马上就会看到，JSON结构由键和值组成，它们可以任意嵌套。JSON类似于其他编程语言中的字典（dictionary）和散列图（hash map）。

② NoSQL这个词出现于2009年，当时很多非关系型数据库越来越流行，NoSQL是它们的总称。

1.2　MongoDB 的主要特性

数据库在很大程度上是由其数据模型来定义的。本节中，我们将了解文档数据模型和MongoDB的特性，这些特性让我们能有效地操作文档数据模型。我们还会看到与运维相关的内容，重点介绍MongoDB的复制和水平伸缩策略。

1.2.1　文档数据模型

MongoDB的数据模型是面向文档的。如果你不熟悉数据库中文档的概念，那我们最好先看一个例子。

代码清单1-1　表示社交新闻网站中一个条目的文档

```
{ _id: ObjectID('4bd9e8e17cefd644108961bb'),    <- _id字段是主键
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
                                                   ❶ 作为字符串数组存储的标签
  tags: ['databases', 'mongodb', 'indexing'],

  image: {                                         ❷ 指向另一文档的属性
    url: 'http://example.com/db.jpg',
    caption: '',
    type: 'jpg',
    size: 75381,
    data: "Binary"
  },
                                                   ❸ 作为评论对象数组存储的评论
  comments: [
    { user: 'bjones',
      text: 'Interesting article!'
    },

    { user: 'blogger',
      text: 'Another related article is at http://example.com/db/db.txt'
    }
  ]
}
```

代码清单1-1是一个示例文档，表示社交新闻网站（比如Digg）上的一篇文章。如你所见，文档基本上是一组属性名和属性值的集合。属性的值可以是简单的数据类型，例如字符串、数字和日期。但这些值也可以是数组，甚至是其他文档❷，这让文档可以表示各种富数据结构。在示例文档中有一个属性`tags`❶，其中用数组的形式保存了文章的标签。更有趣的是`comments`属性❸，它是一个评论文档的数组。

让我们花点时间把它和标准关系型数据库中相同数据的表述对比一下。图1-1是一个对应的关系型数据库的表述。既然数据表本质上来说是扁平的，那么要表示多个一对多关系就需要多张表。先从包含每篇文章核心信息的`posts`表开始，然后创建三张其他的表，每个表都包含一个`post_id`字段指向原始的文章。这种将对象的数据拆分到多张表里的技术称为正规化（normalization）。排

除其他因素，正规化的数据集可以保证每个数据单元仅出现在一个地方。

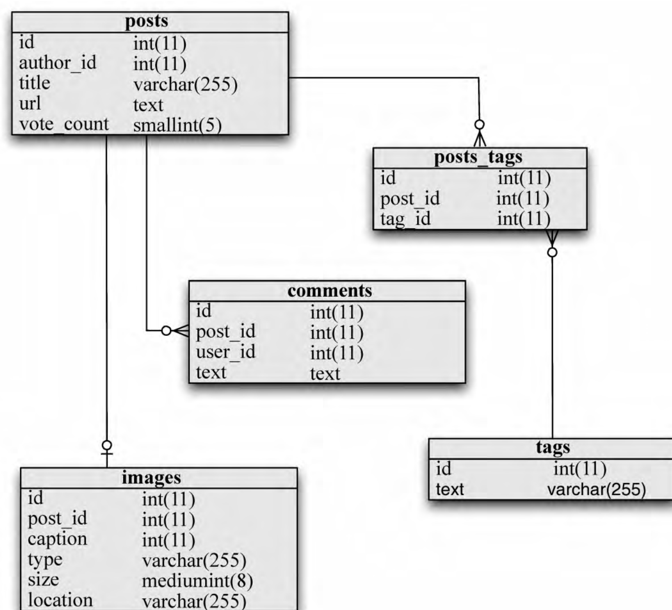


图1-1　表示社交新闻网站中一个条目的基本关系数据模型

但严格的正规化是有代价的，特别是需要一些装配工作。为了显示我们刚刚提到的文章，需要在`posts`和`tags`表之间执行联结操作。还需要单独查询评论，或者也把它们放在一个`join`语句里。最终，是否需要严格正规化要取决于所建模的数据的类型，在第4章我会更深入地讨论这个问题。这里重点说一下，面向文档的数据模型很容易以聚合的形式来表示数据，让你能彻底和对象打交道：所有用来表示一篇文章的数据，从评论到标签，都能放进一个单独的数据库对象里。

你可能已经注意到了，除了提供丰富的结构，文档无需预先定义Schema。在关系型数据库中存储的是数据表中的行，每张表都有严格定义的Schema，规定了列和类型。如果表中的某一行需要一个额外的字段，那么就不得不显式地修改表结构。MongoDB把文档组织成集合，这种容器无需任何类型的Schema。理论上，集合中的每个文档都能拥有完全不同的结构。在实践中，一个集合里的文档相对统一，举例来说，文章集合里的文档都有表示标题、标签、评论等内容的字段。

这种做法带来了一定的优势。首先，是应用程序，而非数据库在保证数据结构。在Schema频繁变化的初期开发阶段，这能提升应用程序的开发效率。其次，更重要的是无Schema的模型允许用真正的可变属性来表示数据。举例来说，假设正在构建一个电子商务产品编目，没办法事先知道产品会有什么属性，因此应用程序需要处理这种可变性。在固定Schema的数据库中，传统的解决方案是使用实体–属性–值模式（entity-attribute-value pattern[①]），如图1-2所示。你所看到的内容

① 参见http://en.wikipedia.org/wiki/Entity-attribute-value_model。

选自Magento的数据模型，这是一个开源的电子商务框架。请注意，这些数据表基本上是一样的，`value`字段除外，该字段仅根据数据类型变化。该结构允许管理员定义附加的产品类型和属性，但却带来了很大的复杂性。试想打开MySQL Shell检查或更新一个用这种方式建模的产品，用于装配该产品的联结语句是何等复杂。以文档的方式建模，就不用做联结，还可以动态地添加新属性。

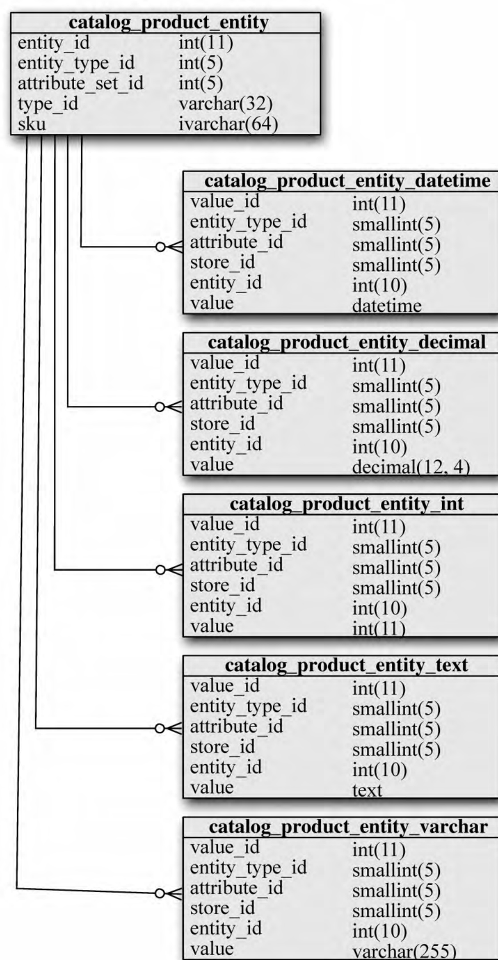


图1-2　PHP电子商务项目Magento的部分Schema，其中这些表用来辅助动态创建产品属性

1.2.2　即时查询

说一个系统支持即时查询（ad hoc query）的意思就是无需预先定义系统接受的查询类型。关系型数据库有这个能力，它们会严格遵照指示执行任何完备的SQL查询，无论有多少条件。如果你仅使用过关系型数据库，那么会认为即时查询是理所应当的。但是，并非所有的数据库都支持

动态查询。举例来说，键值存储只能按一个维度来查询——键。和很多其他系统一样，键值存储牺牲了丰富的查询能力来换取一个简单的可伸缩模型。关系型数据库世界中，查询能力是再基础不过的事情，MongoDB的设计目标之一就是尽可能保留这种能力。

要了解MongoDB的查询语句如何工作，让我们先来看一个简单的例子，它涉及文章和评论。假设想要找到所有带*politics*标签、投票数大于10的文章，SQL查询大概会是这样的：

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id = posts_tags.post_id
  INNER JOIN tags ON posts_tags.tag_id == tags.id
  WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

MongoDB中的等效查询是用文档来做匹配的，特殊的`$gt`键表示“大于”：

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

请注意，这两个查询采用了不同的数据模型。SQL查询依赖于严格正规化的模型，其中文章和标签保存在不同的数据表中，而MongoDB的查询假定标签是存储在每个文章的文档中。两者都演示了对任意属性组合执行查询的能力，这是即时查询的本质。

正如之前提到的，一些数据库的数据模型过于简单，因此不支持即时查询。举例来说，你只能根据主键在键值存储中进行查询。对于查询而言，它并不知道这些键所对应的值。要根据第二属性进行查询，比如本例中的投票数，唯一的方法是自己写代码来构造条目，其中主键是指定的投票数，值是一个文档主键的列表，文档里包含了键中所指定的投票数。如果你在键值存储中使用了这种方法，那么一定会为此而深感愧疚，虽然这种做法在数据集较小时能管用，把多个索引塞进物理结构是单索引的存储中，这并不是一个好主意。而且，键值存储中基于散列的索引不支持范围查询，而在查询类似投票数这样的东西时，范围查询可能是必不可少的。

如果你之前是使用关系型数据库系统的，视即时查询为常态，那么应该会发现MongoDB提供了类似的查询能力。如果正在评估多种不同的数据库技术，请牢记不是所有的数据库都支持即时查询，要是你的确需要这种能力，MongoDB会是一个不错的选择。但光有即时查询是不够的，一旦数据集膨胀到一定程度，出于查询效率就必须使用索引。适当的索引能把查询和排序的速度提升一个数量级，所以支持即时查询的系统还应该要支持二级索引。

1.2.3 二级索引

理解数据库索引的最佳方法就是类比：很多书都有索引，把关键字和页码对应起来。假设你有一本菜谱，想要找到其中要用梨的菜（也许你有很多梨，不想它们坏掉）。最花时间的做法是一页页找过去，看每道菜的配料。大多数人都喜欢查书的索引，从中找到梨那一项，其中会指出所有包含梨的菜。数据库索引就是提供类似服务的数据结构。

MongoDB中的二级索引是用B树（B-tree）实现的，B树索引也是大多数关系型数据库的默认索引，针对多种查询做了优化，包括范围扫描和带排序子句的查询。通过允许使用多个二级索引，MongoDB让用户能对大量不同的查询进行优化。

在MongoDB里，每个集合最多可以创建64个索引。它支持能在RDBMS中找到的各种索引，升序、降序、唯一性、复合键索引，甚至地理空间索引都被支持。因为MongoDB和大多数RDBMS

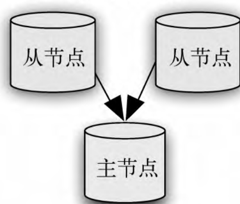
使用相同的索引数据结构，这些系统中有关管理索引的建议都是通用的。下一章里我们会开始介绍索引，因为了解索引对高效操作数据库至关重要，所以我会用整个第7章来讨论这个话题。

1.2.4　复制

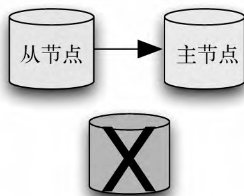
MongoDB通过称为副本集（replica set）的拓扑结构提供了复制功能。副本集将数据分布在多台机器上以实现冗余，在服务器和网络故障时能提供自动故障转移。除此之外，复制功能还能用于扩展数据库的读能力。如果有一个读密集型的应用程序（Web上很常见），可以把数据库读操作分散到副本集集群中的各台机器上。

副本集由一个*主节点*（primary node）和一个或多个*从节点*（secondary node）构成。与你所熟悉的其他数据库中的*主从复制*（master-slave replication）类似，副本集的主节点既能接受读操作又能接受写操作，但从节点是只读的。让副本集与众不同的是它能支持自动故障转移：如果主节点出了问题，集群会选一个从节点自动将它提升为主节点。在先前的主节点恢复之后，它就会变成一个从节点。图1-3描述了这个过程。

1. 一个工作中的副本集



2. 原来的主节点出了问题，一个从节点被提升为主节点



3. 原来的主节点会恢复成一个从节点

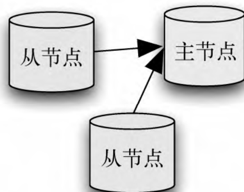


图1-3　副本集的自动故障转移

我会在第8章里详细讨论复制。

1.2.5 速度和持久性

要理解MongoDB实现持久性的方法，需要先理解一些思想。在数据库系统领域内，写速度和持久性存在一种相反的关系。写速度可以理解为在给定时间内数据库可以处理的插入、更新和删除操作的数量。持久性则是指数据库保持这些写操作结果不变的时间长短。

举例来说，假设要向数据库写100条50 KB的记录，随后立即切断服务器的电源。机器重启后这些记录能恢复么？答案是——有可能，这取决于数据库系统和托管它的硬件。问题是写磁盘的速度要比写内存慢几个数量级。某些数据库，例如memcached，只写内存，这让它们速度很快，但数据完全易失。另一方面，几乎没有数据库只写磁盘，因为这样的操作性能过低，无法接受。因此，数据库设计者经常需要在速度和持久性中做出权衡，以平衡两者的关系。

在MongoDB中，用户可以选择写入语义，决定是否开启Journaling日志记录，通过这种方式来控制速度和持久性间的平衡。默认所有的写操作都是fire-and-forget[①]的，即写操作通过TCP套接字发送，不要求数据库应答。如果用户需要获得应答，可以使用特殊的安全模式发起写操作，所有驱动都提供这个安全模式。该模式强制数据库作出应答，确保数据库正确无误地接收到了写操作。安全模式是可配置的，还可用于阻塞操作，直到写操作被复制到特定数量的服务器。对于高容量、低价值的数据（例如点击流和日志），fire-and-forget风格的写操作是很理想的选择。对于重要的数据，则更倾向于安全模式。

在MongoDB 2.0中，Journaling日志是默认开启的。有了这个功能，所有写操作都会被提交到一个只能追加的日志里。即使服务器非正常关闭（比方说电源故障），该日志也能保证在重启服务器后MongoDB的数据文件被恢复到一致的状态。这是运行MongoDB最安全的方式。

事务日志

MySQL的InnoDB中有一个关于速度和持久性的折中。InnoDB是事务性存储引擎，根据定义，必须保证持久性。它通过向两个地方写入更新来实现这一目标：先写事务日志，再写内存缓冲池。事务日志会立刻同步到磁盘，而缓冲池则只会由后台线程最终同步。采取这种双重写入的原因是一般来讲随机I/O要比顺序I/O慢得多。因为向主数据文件的写操作构成随机I/O，所以先写内存会更快，可以后面再同步到磁盘上。但有些写操作（至磁盘）要保证持久性，保证写入是连续的这一点很重要，这就是事务日志的功能。在非正常关闭时，InnoDB能回放事务日志，并依此来更新主数据文件。这种做法在保证高持久性的同时也提供了能接受的性能。

可以在不记日志的情况下运行服务器，这样能提升写入的性能，但在服务器意外关闭后可能会损坏数据文件。其结果就是那些想要关闭Journaling日志功能的人必须使用复制功能，最好还

① 维基百科中解释为“射后不理”，源自军事领域，泛指武器发射后无需外界干涉就能自己更新目标或自己坐标的能力。——译者注

能将数据复制到另一个数据中心，以此来增加失败时还能找回原始数据副本的可能性。

复制和持久性是一个很大的话题，第8章会详细展开讨论的。

1.2.6　数据库扩展

对大多数数据库而言，最简单的扩展方法就是升级硬件。如果应用程序运行在单个节点上，增加磁盘IOPS（Input/Output Operations Per Second，每秒输入输出操作）、内存和CPU通常都可以暂时消除数据库的性能瓶颈。提升单一节点的硬件来进行扩展称为垂直扩展或向上扩展。垂直扩展的优势在于简单、可靠，某种程度上而言还是比较划算的。如果你正在使用虚拟化硬件（比如亚马逊的EC2）上，可能会找不到足够大的实例。如果正在使用物理硬件，终会有一天，更强大的服务器的成本会让你望而却步。

这时就该考虑水平扩展或向外扩展了。水平扩展不是提升单一节点的性能，而是将数据库分布到多台机器上。因为水平扩展架构可以使用普通硬件，所以托管整个数据集的成本会显著降低。而且，将数据分布在多台服务器上可以降低故障带来的影响。有时机器的故障是难以避免的，如果采用的是垂直扩展，在机器发生故障时，你需要处理的就是自己大多数系统所依赖的那台服务器的故障。如果在复制的从服务器上有一份数据副本，问题还不算严重，但在单机故障仍需暂停整个系统时，这依然很棘手。水平扩展架构中的故障与之形成鲜明对比，单节点故障不会带来灾难性影响，因为从整体上看，它只代表了很小一部分数据。图1-4对比了水平扩展和垂直扩展。

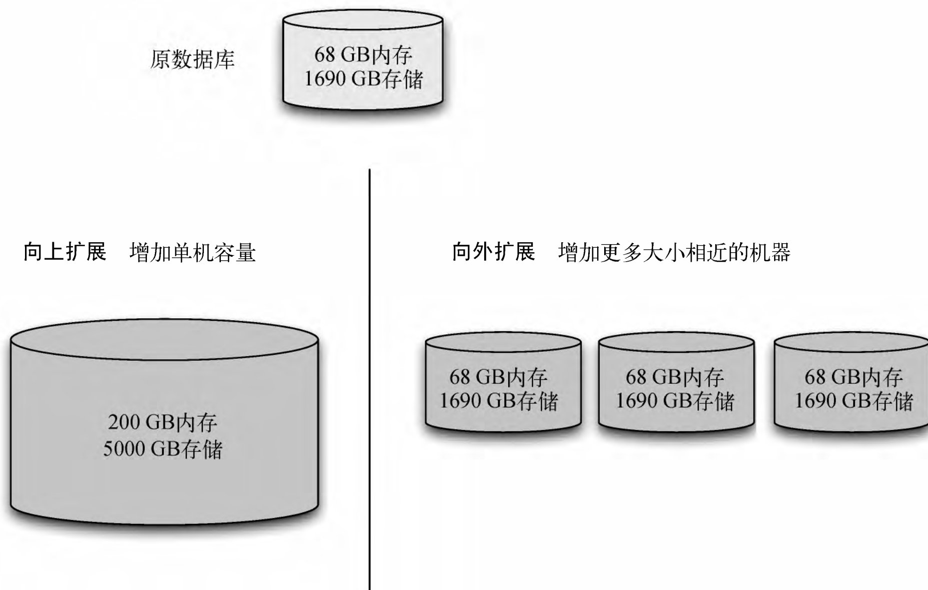


图1-4　水平扩展与垂直扩展

MongoDB的水平扩展非常易于管理，它通过基于范围的分区机制，即自动分片（auto-sharding）来实现这一设计目标，自动分片机制会自动管理各个节点之间的数据分布。分片系统会处理分片节点的增加，帮助进行自动故障转移。单独的分片由一个副本集组成，其中包含至少两个节点[①]，保证能够自动恢复，没有单点失败。综上所述，完全不需要编写应用程序代码来处理这些事情，应用程序的代码只要像和单个节点通信一样来访问分片集群就可以了。

我们已经讲到了MongoDB中大多数的重要特性，第2章将介绍其中一些特性在实践中是如何应用的。但此时此刻，让我们从更实用的角度来看看数据库。MongoDB的核心服务器自带了一套工具，下一节我们将介绍怎么使用这些工具以及一些输入输出数据的方式。

1.3 MongoDB 的核心服务器和工具

MongoDB是用C++编写的，由10gen积极维护。该项目能在所有主流操作系统上编译，包括Mac OS X、Windows和大多数Linux。mongodb.org上提供了这些平台的预编译二进制包。MongoDB是开源的，遵循GNU-AGPL许可，GitHub上可以免费获取到源代码，而且经常会接受来自社区的贡献，但这一项目主要还是由10gen的核心服务器团队来领导的，绝大多数提交亦来自该团队。

GNU-AGPL

GNU-AGPL是一个颇受争议的许可。实践中，它表示源代码能被免费获取，而且它鼓励社区的贡献。GNU-AGPL的主要局限是，出于社区的利益，任何对源代码的修改都必须公布出来。对于那些想保护其核心服务器增强特性的公司来说，10gen提供了特殊的商业许可。

MongoDB 1.0发布于2009年11月。基本每三个月人们便发布它的一个主要版本，偶数发行号[②]代表稳定分支，奇数代表开发分支。在本书编写时，最新版本是v2.0[③]。

下文概述了MongoDB自带的组件，并粗略描述了工具和面向MongoDB开发应用程序所需的语言驱动。

1.3.1 核心服务器

通过可执行文件`mongod`（Windows上是mongodb.exe）可以运行核心服务器。`mongod`服务器进程使用一个自定义的二进制协议从网络套接字上接收命令。`mongod`进程的所有数据文件默认都存储在/data/db[④]里。

`mongod`有多种运行模式，最常见的是作为副本集中的一员。因为推荐使用复制，通常副本集由两个副本组成，再加一个部署在第三台服务器上的仲裁进程[⑤]（arbiter process）。对于MongoDB

① 技术上来看，每个副本集都至少有三个节点，但其中只有两个需要携带数据副本。

② release number，即版本号中的第二个数字。——译者注

③ 你应该总是使用稳定版本，例如2.0.1版。

④ Windows里是c:\data\db。

⑤ 这些仲裁进程都是轻量级的，也就是说能方便地运行在应用服务器上。

的自动分片架构而言，其组件包含配置为预先分片的副本集的mongod进程，以及特殊的元数据服务器，称为配置服务器（config server）。另外还有单独的名为mongos的路由服务器向适当的分片发送请求。

相比其他的数据库系统，例如MySQL，配置一个mongod进程相对比较简单。虽然可以指定标准端口和数据目录，但没有什么调优数据库的选项。在大多数RDBMS中，数据库调优意味着通过一大堆参数来控制内存分配等内容，这已经变成了一门黑魔法。MongoDB的设计哲学指出，内存管理最好是由操作系统而非DBA或应用程序开发者来处理。如此一来，数据文件通过mmap()系统调用被映射成了系统的虚拟内存。这一举措行之有效地将内存管理的重任交给了操作系统内核。本书中我还会更多地阐述与mmap()相关的内容，不过目前你只需要知道缺少配置参数是一个系统设计亮点，而非缺陷。

1.3.2 JavaScript Shell

MongoDB命令行Shell是一个基于JavaScript的工具，用于管理数据库和操作数据。可执行文件mongo会加载Shell并连接到指定的mongod进程。MongoDB Shell的功能和MySQL Shell差不多，主要的区别在于不使用SQL，大多数命令使用的是JavaScript表达式。举例来说，可以像下面这样选择一个数据库，向users集合中插入一个简单的文档：

```
> use mongodb-in-action
> db.users.insert({name: "Kyle"})
```

第一条命令指明了想使用哪个数据库，MySQL的用户一定不会对此感到陌生。第二条命令是一个JavaScript表达式，插入一个简单的文档。要查看插入的结果，可以使用以下查询：

```
> db.users.find()
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name: "Kyle" }
```

find方法返回了之前插入的文档，其中添加了一个对象ID。所有文档都要有一个主键，存储在_id字段里。只要能保证唯一性，也可以输入一个自定义_id。如果省略了_id，则会自动插入一个MongoDB对象ID。

除了可以插入和查询数据，Shell还可以用于运行管理命令。例如，查看当前数据库操作、检查到从节点的复制状态，以及配置一个用于分片的集合。如你所见，MongoDB Shell着实是一个强大的工具，值得好好掌握。

说了这么多，你那些和MongoDB相关的大量工作都是通过特定编程语言编写的应用程序来完成的。想知道这究竟是如何办到的，必须先了解一下MongoDB语言驱动。

1.3.3 数据库驱动

如果之前把数据库驱动想象成捣腾低级设备的梦魇，那你大可放心，MongoDB的驱动很容易使用。MongoDB团队竭尽全力在提供符合特定语言风格的API，并同时保持跨语言的、相对统一的接口。举例来说，所有驱动都实现了向集合保存文档的类似方法，但不同语言里文档本身的表述通常会有所不同，驱动尽量会对特定语言表现得更自然一些。例如，在Ruby中就是使用一个

Ruby散列，在Python中字典更合适一点，Java中缺少类似的语言原语，需要用一个实现了LinkedHashMap的特殊文档构建器类来表示文档。

因为驱动程序为数据库提供了一个以语言为中心的富接口，在构建应用程序时几乎不再需要驱动程序之外的抽象了。这与使用RDBMS的应用程序设计截然不同，在数据库的关系型数据模型和大多数现代编程语言的面向对象模型之间几乎都需要有一个库来做中介。虽然不需要对象关系映射器（object-relational mapper），但很多开发者都喜欢在驱动上做一层薄薄的封装，用它来处理关联、验证和类型检查[①]。

本书编写时，10gen官方支持C、C++、C#、Erlang、Haskell、Java、Perl、PHP、Python、Scala和Ruby的驱动，而且这个列表还在不断增长。如果你需要支持其他语言，通常都会有一个社区支持的驱动。如果对于某语言还没有社区支持的驱动，mongodb.org的文档里有用于构建新驱动的规范。官方支持的驱动被大量使用在生产环境中，而且这些驱动都遵循Apache许可，因此想要编写驱动的人可以免费获取到大量优秀的示例。

从第3章开始，我会描述驱动是如何工作的，以及如何使用它们编写程序。

1.3.4 命令行工具

MongoDB自带了很多命令行工具。

- mongodump和mongorestore，备份和恢复数据库的标准工具。mongodump用原生的BSON格式将数据库的数据保存下来，因此最好只是用来做备份，其优势是热备时非常有用，备份后能方便地用mongorestore恢复。
- mongoexport和mongoimport，用来导入导出JSON、CSV和TSV数据，数据需要支持多种格式时很有用。mongoimport还能用于大数据集的初始导入，但是在导入前顺便还要注意一下，为了能充分利用好MongoDB通常需要对数据模型做些调整。在这种情况下，通过使用驱动的自定义脚本来导入数据会更方便一些。
- mongosniff，这是一个网络嗅探工具，用来观察发送到数据库的操作。基本就是把网络上传输的BSON转换为易于人们阅读的Shell语句。
- mongostat，与iostat类似，持续轮询MongoDB和系统以便提供有帮助的统计信息，包括每秒操作数（插入、查询、更新、删除等）、分配的虚拟内存数量以及服务器的连接数。

稍后会在书中讨论另外两个工具：bsondump和mongofiles。

1.4 为什么选择 MongoDB

为什么MongoDB对于你的项目来说是一个好的选择？我想我已经给出不少理由了。本节中，我会更明确地进行说明，首先说说MongoDB项目的总体设计目标。根据其作者的观点，MongoDB的设计是要结合键值存储和关系型数据库的最好特性。键值存储，因为非常简单，所以速度极快

① 在本书编写时，一些流行的包装器包括Java的Morphia、PHP的Doctrine以及Ruby的MongoMapper。

而且相对容易伸缩。关系型数据库较难伸缩，至少很难水平伸缩，但拥有富数据模型和强大的查询语言。如果MongoDB能介于两者之间，就能成为一款易伸缩、能存储丰富数据结构、提供复杂查询机制的数据库。

在使用场景方面，MongoDB非常适合用做以下应用程序的主要数据存储：Web应用程序、分析与记录应用程序，以及任何要求有中等级别缓存的应用程序。此外，由于它能方便地存储无Schema数据，MongoDB还很适合保存事先无法知晓其数据结构的数据。

之前所说的内容还不太足以让人信服，为了证实它们，我们大致了解一下目前市面上的众多数据库，并和MongoDB做个对比。接下来，我将讨论一些特殊的MongoDB使用场景，提供一些生产环境中的例子。最后，我还会讨论一些MongoDB实际使用中的重要注意事项。

1.4.1　MongoDB与其他数据库的对比

市面上的数据库数量成爆炸式增长，要在它们之间进行权衡是很困难的。幸运的是，它们之中的大多数数据库都能归在几个分类里。本节中，我会描述简单及复杂的键值存储、关系型数据库和文档数据库，并将它们与MongoDB做一个比较。下面来看表1-1。

表1-1　数据库家族

	示　　例	数据模型	伸缩性模型	使用场景
简单键值存储	memcached	键值对，其中值是一个二进制大字段	多种模型。memcached能跨多个节点进行伸缩，把所有可用内存变为一个巨大的数据存储	缓存、Web操作
复杂键值存储	Cassandra、Project Voldemort、Riak	多种模型。Cassandra使用名为列（column）的键值结构。Voldemort存储二进制大字段	最终一致性，多节点部署以获得高可用性和简单的故障转移	高吞吐量垂直内容（活动feed、消息队列）、缓存、Web操作
关系型数据库	Oracle数据库、MySQL、PostgreSQL	数据表	垂直伸缩。对集群和手动分区支持有限	要求事务（银行、金融）或SQL的系统、正规化数据模型

1. 简单键值存储

简单键值存储正如其名，基于给定的键对值做索引。常见的场景是缓存。举例来说，假设需要缓存一个由应用程序呈现的HTML页面，此处的键可能是页面的URL，值是HTML本身。请注意，对键值存储而言，值就是一个不透明的字节数组。没有强加关系型数据库中的Schema，也没有任何数据类型的概念。这自然限制了键值存储允许的操作：可以放入一个新值，然后通过键将其找出或删除。拥有如此简单性的系统通常很快，而且具有可伸缩性。

最著名的简单键值存储是memcached（发音是*mem-cash-dee*）。memcached仅在内存里存储数据，用持久性来换取速度。它也是分布式的，跨多台服务器的memcached节点能像单个数据存储那样来使用，这消除了维护跨服务器缓存状态的复杂性。

与MongoDB相比，memcached这样的简单键值存储通常读写会更快。但与MongoDB不同，这些系统很少能充当主要数据存储。简单键值存储的最佳用途是附加存储，既可以作为传统数据库之上的缓存层，也可以作为任务队列之类的短暂服务的简单持久层。

2. 复杂键值存储

可以改进简单键值模型来处理复杂的读写Schema或提供更丰富的数据模型。如此一来，就有了复杂键值存储。广为流传的论文“Dynamo: Amazon’s Highly Available Key-value Store”中描述的亚马逊 Dynamo就是这样一个例子。Dynamo旨在成为一个健壮的数据库，在网络故障、数据中心停转及类似情况下仍能工作。这要求系统总是能够被读和写，本质上就是要求数据能自动跨多个节点进行复制。如果一个节点发生故障，系统的用户（在这里可能是一个使用亚马逊购物车的顾客）不会察觉到服务中断。当系统允许同一份数据被写到多个节点时，发生冲突的情况是不可避免的，Dynamo提供了一些解决冲突的方法。与此同时，Dynamo也很容易伸缩。因为没有主节点，所有节点都是对等的，所以很容易从整体上理解系统，能方便地添加节点。尽管Dynamo是一个私有系统，但其构建理念启发了很多NoSQL系统，包括Cassandra、Project Voldemort和Riak。

看看是谁开发了这些复杂键值存储，看看实践中它们的使用情况如何，你就能知道它们的优点了。以Cassandra为例，它实现了很多Dynamo的伸缩属性，同时还提供了与谷歌 BigTable类似的面向列的数据模型。Cassandra是一款开源的数据存储，是Facebook为其收件箱搜索功能开发的。该系统可以水平扩展，索引超过50 TB的收件箱数据，允许在收件箱中对关键字和收件人做检索。数据是根据用户ID做索引的，每条记录由一个用于关键字检索的搜索项数组和一个用于收件人检索的收件人ID数组构成。[①]

这些复杂键值存储是由亚马逊、谷歌和Facebook这样的大型互联网公司开发的，用来管理系统的多个部分，拥有非常大的数据量。换言之，复杂键值存储管理了一个相对自包含的域，它对海量存储和可用性有一定要求。由于采用了无主节点的架构，这些系统能轻松地通过添加节点进行扩展。它们都选择了最终一致性，也就是说读请求不必返回最后一次写的内容。用户用较弱的一致性所换得的是在某一节点失效时仍能写入的能力。

这与MongoDB正好相反，MongoDB提供了强一致性、（每个分片）一个主节点、更丰富的数据模型，还有二级索引，最后两项特性总是一起出现的。如果一个系统允许跨多个域建模，例如构建完整Web应用程序时就会有此要求，那么查询就需要跨整个数据模型，这时就要用到二级索引了。

因为有丰富的数据模型，可以考虑把MongoDB作为更通用的大型、可伸缩Web应用程序的解决方案。MongoDB的伸缩架构有时也会受到非难，因为它并非源自Dynamo。但MongoDB针对不同域有不同的伸缩解决方案。MongoDB的自动分片受到了雅虎PNUTS数据存储和谷歌 BigTable的启发。读过发布这些数据存储的白皮书的人会发现，MongoDB实现伸缩的方法已经被实现了，而且还很成功。

① 参见http://mng.bz/5321。

3. 关系型数据库

本章已经介绍了不少关系型数据库的内容，简单起见，我只讨论RDBMS与MongoDB的相同点和不同点。尽管MySQL[1]使用固定Schema的数据表，MongoDB使用无Schema的文档，但两者都能表示丰富的数据模型。MySQL和MongoDB都支持B树索引，那些适用于MySQL索引的经验也同样适用于MongoDB。MySQL支持联结和事务，因此如果你必须使用SQL或者要求有事务，那么只能选择MySQL或其他RDBMS。也就是说，MongoDB的文档模型足以在不用联结查询的情况下表示对象。MongoDB中对单独文档的更新也是原子的，这提供了传统事务的一个子集。MongoDB和MySQL都支持复制。就可伸缩性而言，MongoDB设计成能水平扩展，能自动分片并处理故障转移。MySQL上的分片都需要手动管理，有一定的复杂性，更常见的是垂直扩展的MySQL系统。

4. 文档数据库

自称为文档数据库的产品还不多，在本书编写时，除了MongoDB之外，唯一的著名文档型数据库就是Apache CouchDB。尽管CouchDB的数据是使用JSON格式的纯文本存储的，而MongoDB是使用BSON二进制格式，但两者的文档模型是相似的。与MongoDB一样，CouchDB也支持二级索引，不同之处是CouchDB中的索引是通过编写MapReduce函数来定义的，这比MySQL和MongoDB使用的声明式语法更复杂一些。两者伸缩的方式也有所不同，CouchDB不会把数据分散到多台服务器上，每个CouchDB节点都是其他节点的完整副本。

1.4.2　使用场景和生产部署

老实说，我们不会仅根据数据库的特性做选择，还需要知道使用它的真实成功案例。这里，我提供一些广义上的MongoDB使用场景，以及一些生产环境中的示例[2]。

1. Web应用程序

MongoDB很适合作为Web应用程序的主要数据存储。就算是一个简单的Web应用程序也会有很多数据模型，用来管理用户、会话、应用特定的数据、上传和权限，更不用说非常重要的域了。正如它们能和关系型数据库的表列数据配合良好一样，它们也能获益于MongoDB的集合与文档模型。因为文档能表示丰富的数据结构，建模相同数据所需的集合数量通常会比使用完全正规化关系型模型的数据表数量要少。此外，动态查询和二级索引能让你轻松地实现SQL开发者所熟悉的大多数查询。最后，作为一个成长中的Web应用程序，MongoDB提供了清晰的扩展路线。

在生产环境中，MongoDB已经证明它能管理应用的方方面面，从主要数据领域到附加数据存储，比如日志和实时分析。这里的案例来自The Business Insider（TBI），它从2008年1月起将MongoDB作为主要数据存储使用。虽然TBI是一个新闻网站，但它流量很大，每天有超过一百万独立页面访问。这个案例中有意思的是除了处理站点的主要内容（文章、评论、用户等），MongoDB还处理并存储实时分析数据。这些分析被TBI用于生成动态热点地图，标明不同新闻故事的点击

---

① 这里我用MySQL来做说明，因为我所描述的特性适用于大多数关系型数据库。

② 要想获得在生产环境中使用了MongoDB的最新案例列表，请访问http://mng.bz/z2CH。

率。该站目前还没有太多的数据，因此尚不需要分片，但它有使用副本集来保证停电时的自动故障转移。

2. 敏捷开发

无论如何看待敏捷开发运动，你都很难否认对于快速构建应用程序的渴望。不少开发团队，包括Shutterfly和纽约时报的团队，都部分选择了MongoDB，因为相比关系型数据库，使用MongoDB能更快地开发应用程序。一个明显的原因是MongoDB没有固定的Schema，所有花在提交、沟通和实施Schema变更的时间都省下来了。

除此之外，不再需要花时间把数据的关系型表述硬塞进面向对象的数据模型里去了，也不用处理ORM生成的SQL的奇怪行为，或者对它做优化了。如此一来，MongoDB为项目带来了更短的开发周期和敏捷的、中等大小的团队。

3. 分析和日志

我之前已经暗示过MongoDB适用于分析和日志，将MongoDB用于这些方面的应用程序数量增长得很快。通常，发展成熟的公司都会选择将用于分析的特殊应用作为切入点，进入MongoDB的世界。这些公司包括GitHub、Disqus、Justin.tv和Gilt Groupe，还有其他公司就不再列举了。

MongoDB与分析的关联源自于它的速度和两个关键特性：目标原子更新和固定集合（capped collection）。原子更新让客户端能高效地增加计数器，将值放入数组。固定集合，常用于日志，特点是分配的大小固定，能实现自动过期。相比文件系统，将日志数据保存在数据库里更易组织，而且能提供更强大的查询能力。现在，抛开grep或自定义日志检索工具，用户可以使用他们熟悉并喜欢的MongoDB查询语言来查看日志输出。

4. 缓存

这是一种数据模型，它能更完整地表示对象，结合更快的平均查询速度，经常让MongoDB介于传统的MySQL与memcached之间。例如之前提到的TBI，它可以不使用memcached，直接通过MongoDB来响应页面请求。

5. 可变Schema

看看这段代码示例[①]：

```
curl https://stream.twitter.com/1/statuses/sample.json -umongodb:secret
| mongoimport -c tweets
```

这里从Twitter流上拉下一小段示例，并用管道将其直接导入MongoDB集合。因为流生成的是JSON文档，在把它发给数据库前就不需要预先处理数据了。mongoimport工具能直接将数据转换成BSON。这意味着每条推文都能保持其结构，原封不动地存储为集合中的单个文档。无论你是想查询、索引还是执行MapReduce聚合，都能立刻操作数据。而且，不需要事先声明数据的结构。

如果应用程序需要调用JSON API，那么拥有这样一个能轻松转换JSON的系统就太棒了。如

① 这个想法来自于http://mng.bz/52XI。如果想运行这段代码，需要把-umongodb:secret替换成自己的Twitter用户名和密码。

果在存储之前无法预先了解数据的结构，MongoDB没有Schema约束的特性能大大简化数据模型。

1.5　提示与局限

有了这些优良的特性，你还需要牢记系统舍弃的特性与局限。在使用MonogDB构建真实的应用程序并用于生产环境之前，应该注意一些局限，它们大多数都是由于MongoDB使用内存映射文件（memory-mapped file）而导致的。

首先，MongoDB通常应该运行于64位的机器上。32位系统只能对4 GB内存做寻址。要知道，一般半数的内存都会被操作系统和程序进程占用，就只剩2 GB内存能用来映射数据文件。因此，如果运行在32位的服务器上，还定义了适当数量的索引，那么数据文件只能被限制在1.5 GB。大多数生产环境系统的要求都高很多，因此一个64位的系统是必需的。[①]

使用虚拟内存映射的第二个后果是，数据占用的内存会自动按需分配。这样一来，想在共享环境中运行数据库会变得更加麻烦。要把MongoDB用于数据库服务器，最好是能让它运行在一台专门的服务器上。

最后，运行带复制功能的MongoDB是十分重要的，尤其是没有开启Journaling日志的时候。因为MongoDB使用了内存映射文件，不开启Journaling日志的话，`mongod`发生任何意外关闭都会导致数据损坏。因此，这时最好能有一个备份以做故障转移。对任何数据库而言这都是个不错的建议（重要的MySQL部署不做复制是很轻率的举动），这对没有Journaling日志的MongoDB尤为重要。

1.6　小结

本章我们讲述了很多内容。概括一下，MongoDB是一款开源的、基于文档的数据库管理系统，是针对现代互联网应用程序的数据和伸缩性要求而设计的，其特性包括动态查询、二级缓存、快速的原子更新和复杂的聚合，还支持基于自动故障转移的复制和用于水平扩展的自动分片。

说了这么多，你应该对这些功能都有了较好的了解。也许你对编码已经跃跃欲试了，毕竟讨论数据库的特性是一回事，在实践中使用数据库又是另一回事。接下来的两章我们就来实践一下。首先，你将接触到MongoDB JavaScript Shell，在与数据库交互时它太有用了。接下来，第3章将带你学习使用驱动，用Ruby构建一个简单的基于MongoDB的应用程序。

---

① 理论上来说，64位的架构可以寻址16艾字节（exabyte）内存，无论想做什么都不会再受到限制。

第 2 章 MongoDB JavaScript Shell

本章内容

- MongoDB Shell 中的 CRUD 操作
- 构建索引与使用 `explain()`
- 获得帮助

上一章里介绍了一些MongoDB的使用经验，本章则给出了更实用的入门知识。通过MongoDB Shell，本章用一系列练习讲述了该数据库的基本概念。你将了解到如何创建、读取、更新、删除（CRUD）文档，且在此过程中还能了解MongoDB的查询语言。除此之外，我们还会大致了解一下数据库索引，以及如何用它们来优化查询。本章结尾给出了一些基本的管理命令，以及有关如何获得帮助的建议。你可以把本章看做对已经介绍过的概念的一个细化，也可以看做是对MongoDB Shell中常用任务的一个实用介绍。

如果是第一次接触MongoDB Shell，那么请记住它能满足你对此类工具的所有期望，它允许查看并操作数据以及管理数据库服务器。它与同类工具的不同之处在于查询语言，它没有使用SQL这样的标准化查询语言，而是使用JavaScript编程语言和一套简单的API与服务器交互。如果你不熟悉JavaScript，只需了解最基本的语言知识就能使用MongoDB Shell了。本章中所有的示例都会有详细的说明。

如果你照着例子学习本章的内容，一定能事半功倍，为此你需要在自己的电脑上安装MongoDB（附录A中有安装指南）。

2.1 深入 MongoDB Shell

MongoDB JavaScript Shell能让你玩转数据，对文档、集合以及MongoDB的特殊查询语言有切实的体验。你可以把以下内容当做对MongoDB的实用入门。

我们先说说Shell的启动与运行，然后再看看JavaScript是如何表示文档并将其插入MongoDB集合的。为了验证插入是否成功，你可以查询集合内容。紧随其后的是更新集合。最后，你还将了解到如何清除并删除集合。

2.1.1　启动 Shell

如果已按照附录A的说明进行了安装，那么电脑上现在应该已经有一个可正常工作的MongoDB了。请确保有一个正在运行的mongod实例，随后运行可执行文件mongo启动MongoDB shell：

```
./mongo
```

如果Shell程序启动成功，你将看到如图2-1所示的界面。Shell开始的地方显示了正运行的MongoDB版本，还有和当前选中的数据库相关的一些信息。



图2-1　启动后的MongoDB JavaScript Shell

如果你懂点JavaScript，马上就可以键入代码使用Shell。如果不懂，就请继续读下去，看看如何插入自己的第一条数据。

2.1.2　插入与查询

如果启动时没有指定其他数据库，Shell会选择名为test的默认数据库。为了让后续的教学练习都在同一个命名空间里，我们先切换到tutorial数据库：

```
> use tutorial
switched to db tutorial
```

你会看到一行消息，说明你已经切换了数据库。

创建数据库与集合

你也许会感到奇怪：我们并没有创建tutorial数据库，又怎么能切换过去呢？实际上，创建数据库并不是必需的操作。数据库与集合只有在第一次插入文档时才会被创建。这个行为与MongoDB对数据的动态处理方式是一致的；因为不用事先定义文档的结构，单独的集合和数据库可以在运行时才被创建。这能简化并加速开发过程，而且有利于动态分配命名空间，很多时候这都很管用。如果你担心数据库或集合被意外创建，大多数驱动都能开启严格模式（strict mode），避免此类由疏忽引起的错误。

现在是时候创建你的第一个文档了。因为正在使用JavaScript Shell，所以将用JSON（JavaScript Object Notation）来描述文档。举个例子，一个最简单的描述用户的文档如下：

```
{username: "jones"}
```

该文档包含一对键和值，存储了Jones的用户名。要保存这个文档，需要选择一个集合，像下面这样把它保存到users集合里就再恰当不过了：

```
> db.users.insert({username: "smith"})
```

在键入这行代码后你会感觉到一丝延迟。这时tutorial数据库和users集合都还没在磁盘上创建出来，延迟是因为要为它们的初始化数据文件分配空间。

如果插入成功，那么就已经成功地保存了第一个文档。可以通过一条简单的查询来进行验证：

```
> db.users.find()
```

查询的结果看起来是这样的：

```
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
```

请注意，文档中添加了_id字段，你可以把它当做文档的主键。每个MongoDB文档都要求有一个_id，如果文档创建时没有提供该字段，就会生成一个特殊的MongoDB对象ID并添加到文档里。在你——控制台里出现的对象ID与示例中的并不一样，但它在集合的所有_id值里是唯一的，这是对该字段的唯一硬性要求。

在下一章里我会详细介绍对象ID。现在继续向集合中添加用户：

```
> db.users.save({username: "jones"})
```

集合里现在应该有两个文档了。接下来通过count命令验证一下：

```
> db.users.count()
2
```

既然集合里文档的数量已经不止一个了，那么就能看些稍微复杂一点儿的查询了。与之前一样，可以把集合里所有的文档都查出来：

```
> db.users.find()
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "smith" }
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

但也可以给find方法传入一个简单的查询选择器。查询选择器（query selector）是一个文档，用来和集合中所有的文档进行匹配。要查询所有用户名是jones的文档，可以像下面这样传入一个简单的文档，将其作为查询选择器：

```
> db.users.find({username: "jones"})
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "jones" }
```

查询选择器{username: "jones"}返回了所有用户名是jones的文档——它会逐字匹配现有的文档。

我刚才演示了创建和读取数据的基本方法，现在再来看看如何更新数据。

2.1.3　更新文档

所有的更新操作都要求至少有两个参数，第一个指明要更新的文档，第二个定义被选中的文档应该如何更新。有两种风格的更新；本节只关注针对性更新（targeted modification），这是MongoDB独有特性中最具代表性的。

举例来说，假设用户smith想要向自己的住所中添加国家信息，可以使用如下更新语句：

```
> db.users.update({username: "smith"}, {$set: {country: "Canada"}})
```

这条更新语句告诉MongoDB应找到用户名是smith的文档，将其country属性值设置为Canada。如果现在执行查询，你将看到更新后的文档：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("4bf9ec440e32f82523389316"),
  "country" : "Canada", username : "smith" }
```

稍后，如果用户决定档案中不再保留国家信息，使用$unset操作符就能轻松去除该值：

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
```

让我们再丰富一下这个例子。如第1章所示，你使用文档来表示数据，其中能包含复杂的数据结构。因此，让我们假设一下，除了存储个人档案信息，用户还能用列表来存储自己喜欢的东西。一个好的文档表述看起来可能是这样的：

```
{ username: "smith",
  favorites: {
    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
  }
}
```

favorites键指向一个对象，后者包含两个其他的键，它们分别指向喜欢的城市列表和电影列表。就目前所知道的知识，你能否想出一个办法将原来的smith文档修改成这样？你应该能想到$set操作符。请注意，本例实际是在改写文档，这也是$set的合理用法：

```
> db.users.update( {username: "smith"},
{ $set: {favorites:
    {
      cities: ["Chicago", "Cheyenne"],
      movies: ["Casablanca", "The Sting"]
    }
}
})
```

让我们对jones做类似修改，但这里就添加两部喜欢的电影：

```
db.users.update( {username: "jones"},
  {"$set": {favorites:
    {
       movies: ["Casablanca", "Rocky"]
    }
  }
})
```

现在查询users集合，确保两个更新都成功了：

```
> db.users.find()
```

有了之前的几个示例文档，现在可以一窥MongoDB查询语言的威力了。尤其值得一提的是，它的查询引擎能深入内嵌对象，匹配数组元素，这种情况下这种能力特别有用。你可以使用特殊的点符号来实现这类查询。假设想找到所有喜欢电影《卡萨布兰卡》（*Casablanca*）的用户，能用这样的查询：


```
> db.users.find({"favorites.movies": "Casablanca"})
```

favorites和movies之间的点告诉查询引擎应找一个名为favorites的键，它指向一个对象（该对象有一个名为movies的内部键），然后匹配它的值。这条查询会把两个用户文档都返回。更进一步，假设你知道每个喜欢《卡萨布兰卡》的用户都喜欢《马耳他之鹰》（*The Maltese Falcon*），且想更新数据库来反映这个情况，该如何用一条MongoDB的更新语句来表示呢？

你可以再次请出$set操作符，但这要求改写并发送整个movies数组。既然你想做的只是向列表里添加一个元素，最好使用$push或$addToSet，这两个操作符都是向数组中添加一个元素，但后者会保证唯一性，防止重复添加。下面就是你要找的更新语句：

```
db.users.update( {"favorites.movies": "Casablanca"},
    {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
          false,
          true )
```

这条语句大体上还是易懂的：第一个参数是一个查询选择器，匹配电影列表里有*Casablanca*的用户；第二个参数使用$addToSet操作符向列表中添加了*The Maltese Falcon*；第三个参数false现在暂时忽略；第四个参数true说明这是一个*多项更新*（multi-update）。MongoDB的更新操作默认只会应用于查询选择器匹配到的第一个文档。如果希望操作被应用于匹配到的所有文档，需要显式说明。因为你希望对smith和jones都进行更新，所以多项更新是必需的。

我们稍后会对更新做更详细的说明，但请先试试这些例子。

2.1.4 删除数据

你已经知道了在MongoDB Shell中创建、读取和更新数据的基本方法了，最后我们来看最简单的操作——删除数据。

如果不加参数，删除操作会清空集合。要干掉foo集合，只需键入：

```
> db.foo.remove()
```

通常只需要删除集合文档的一个子集，为此可以给remove()方法传入一个查询选择器。如果想要删除所有喜欢城市Cheyenne的用户，用下面这个表达式就行了：

```
> db.users.remove({"favorites.cities": "Cheyenne"})
```

请注意，remove()操作不会删除集合，它只是从集合中删除文档。你可以把它想象成SQL中的DELETE和TRUNCATE TABLE指令。

如果想删除集合以及它的全部索引，可以使用drop()方法：

```
> db.users.drop()
```

创建、读取、更新和删除是所有数据库的基本操作。如果你读过了前面的内容，现在应该能在MongoDB中实践这些基本的CRUD操作了。在下一节里，你将了解到二级索引，通过它来提高查询、更新和删除的性能。

2.2　创建索引并查询

创建索引来提升查询性能是很常见的做法。很幸运，你能轻松地在Shell中创建MongoDB的索引。如果没接触过数据库索引，本节内容会让你理解对它们的需求；如果有过索引的使用经验，你会发现创建索引然后使用explain()方法根据索引来剖析查询有多么方便。

2.2.1　创建一个大集合

只有集合中的文档达到一定的数量之后，索引示例才有意义。因此，向numbers集合中添加200 000个简单文档。因为MongoDB Shell也是一个JavaScript解释器，所以实现这一功能的代码很简单：

```
for(i=0; i<200000; i++) {
  db.numbers.save({num: i});
}
```

这些文档数量不少，因此如果插入花了不少时间也不用感到惊讶。执行返回后，可以运行两条查询来验证文档全部存在：

```
> db.numbers.count()
200000

> db.numbers.find()
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830a"), "num" : 0 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830b"), "num" : 1 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830c"), "num" : 2 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830d"), "num" : 3 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830e"), "num" : 4 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac830f"), "num" : 5 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8310"), "num" : 6 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8311"), "num" : 7 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8312"), "num" : 8 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8313"), "num" : 9 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8314"), "num" : 10 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8315"), "num" : 11 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8316"), "num" : 12 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8317"), "num" : 13 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8318"), "num" : 14 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8319"), "num" : 15 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831a"), "num" : 16 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831b"), "num" : 17 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831c"), "num" : 18 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831d"), "num" : 19 }
has more
```

count()命令说明插入了200 000个文档，随后的查询显示了前20个结果，你可以用it命令显示更多查询结果：

```
> it
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831e"), "num" : 20 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
...
```

it命令会告诉Shell返回下一个结果集。[1]

手头有了数量可观的文档之后，我们试着运行一些查询。就你目前对MongoDB查询引擎的了解，一个简单的匹配num属性的查询很好理解：

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac84fe"), "num" : 500 }
```

但更值得一提的是，你还可以使用特殊的$gt和$lt操作符（最早见于第1章，分别表示大于和小于）来执行范围查询。下面的语句用来查询num值大于199 995的所有文档：

```
> db.numbers.find( {num: {"$gt": 199995 }} )
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcade"), "num" : 199996 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcadf"), "num" : 199997 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae0"), "num" : 199998 }
{ "_id" : ObjectId("4bfbf1dedba1aa7c30afcae1"), "num" : 199999 }
```

还可以结合使用这两个操作符指定上界和下界：

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac831f"), "num" : 21 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8320"), "num" : 22 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8321"), "num" : 23 }
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac8322"), "num" : 24 }
```

可以看到，使用简单的JSON文档，可以像在SQL中一样声明复杂的范围查询。MongoDB查询语言由大量特殊关键字组成，$gt和$lt只是其中的两个，在后续的章节中你还会看到更多查询的例子。

当然，这样的查询如果效率不高，那么几乎一点儿价值都没有。下一节中我们将探索MongoDB的索引特性，开始思考查询效率。

2.2.2 索引与 explain()

如果你使用过关系型数据库，想必对SQL的EXPLAIN并不陌生。EXPLAIN用来描述查询路径，通过判断查询使用了哪个索引来帮助开发者诊断慢查询。MongoDB也有提供相同服务的“EXPLAIN”。为了了解它是如何工作的，先在运行过的查询上试一下：

---

① 你也许想知道背后究竟发生了什么。所有的查询都会创建一个游标，可以迭代结果集。这个过程是隐藏在Shell的使用过程中的，因此目前还没有必要详细说明。如果你迫不及待地想深入了解游标及其特性，可以阅读第3章和第4章。

```
> db.numbers.find( {num: {"$gt": 199995 }} ).explain()
```

返回结果如代码清单2-1所示。

代码清单2-1　无索引查询的典型explain()输出

```
{
  "cursor" : "BasicCursor",
  "nscanned" : 200000,
  "nscannedObjects" : 200000,
  "n" : 4,
  "millis" : 171,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : { }
}
```

查看explain()的输出，你会惊讶地发现，查询引擎为了返回4个结果（n）扫描了整个集合，即全部200 000个文档（nscanned）。BasicCursor游标类型说明该查询在返回结果集时没有使用索引。扫描文档和返回文档数量之间巨大的差异说明这是一个低效查询。在现实当中，集合与文档本身可能会更大，处理查询所需的时间将大大超过此处的171 ms。

这个集合需要索引。你可以通过ensureIndex()方法为num键创建一个索引。请输入下列索引创建代码：

```
> db.numbers.ensureIndex({num: 1})
```

与查询和更新等其他MongoDB操作一样，你为ensureIndex()方法传入了一个文档，定义索引的键。这里，文档{num:1}说明为numbers集合中所有文档的num键构建一个升序索引。

可以调用getIndexes()方法来验证索引是否已经创建好了：

```
> db.numbers.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "tutorial.numbers",
    "key" : {
      "_id" : 1
    }
  },

  {
    "_id" : ObjectId("4bfc646b2f95a56b5581efd3"),
    "ns" : "tutorial.numbers",
    "key" : {
      "num" : 1
    },
    "name" : "num_1"
  }
]
```

该集合现在有两个索引了，第一个是为每个集合自动创建的标准_id索引，第二个是刚才在num上创建的索引。

如果现在再来运行explain()方法，在查询的响应时间上会有巨大的差异，如代码清单2-2所示。

代码清单2-2　有索引查询的explain()输出

```
> db.numbers.find({num: {"$gt": 199995 }}).explain()
{
  "cursor" : "BtreeCursor num_1",
  "indexBounds" : [
    [
      {
        "num" : 199995
      },
      {
        "num" : 1.7976931348623157e+308
      }
    ]
  ],
  "nscanned" : 5,
  "nscannedObjects" : 4,
  "n" : 4,
  "millis" : 0
}
```

现在查询利用了num上的索引，只扫描了5个文档，将查询时间从171 ms降到了1 ms以下。

如果这个例子激起了你的兴趣，请不要错过专门介绍索引和查询优化的第7章。接下来让我们看看基本的管理命令，它们可以用来获取MongoDB实例的信息。你还将了解到一些技术，它们与如何在Shell里获取帮助相关，这有助于掌握众多Shell命令。

2.3　基本管理

本章承诺过要通过JavaScript Shell来介绍MongoDB。你已经了解了数据操作和索引的基本知识，这里我将介绍一些技术，帮助你获得mongod进程的信息。举例来说，你可能想知道众多集合一共占用了多少空间，或者你在一个集合上定义了多少索引。此处详述的命令能帮助你诊断性能问题并监控数据。

我们还会了解MongoDB的命令界面。大多数能在MongoDB实例上执行的特殊非CRUD操作——从服务器状态检查到数据文件完整性校验——都是由数据库命令实现的。我将说明MongoDB上下文里的命令，并演示其易用性。最后，知道如何寻求帮助总是好的，所以我将指出在Shell里怎么获得帮助，以帮助你进一步了解MongoDB。

2.3.1　获取数据库信息

你经常想知道指定实例上到底有哪些集合与数据库，幸运的是MongoDB Shell提供了许多命

令和语法糖，以此能获取系统相关的信息。

show dbs显示了系统上所有数据库的列表：

```
> show dbs
admin
local
test
tutorial
```

show collections显示了定义在当前数据库里的所有集合的列表。[①]如果目前还是选中了tutorial数据库，你会看到之前用过的集合：

```
> show collections
numbers
system.indexes
users
```

你可能会对其中的system.indexes集合感到陌生。这是存在于每个数据库中的特殊集合，其中每一项都定义了数据库的一个索引。我们能直接查询该集合，但这样的输出不易阅读，还是像我们之前看到的那样，使用getIndexes()方法更好一些。

为了获得数据库与集合更底层的信息，stats()方法非常有用。在数据库对象上执行该方法时，会获得如下输出：

```
> db.stats()
{
  "collections" : 4,
  "objects" : 200012,
  "dataSize" : 7200832,
  "storageSize" : 21258496,
  "numExtents" : 11,
  "indexes" : 3,
  "indexSize" : 27992064,
  "ok" : 1
}
```

我们还可以在单独的集合上执行stats()命令：

```
> db.numbers.stats()
{
  "ns" : "tutorial.numbers",
  "count" : 200000,
  "size" : 7200000,
  "storageSize" : 21250304,
  "numExtents" : 8,
  "nindexes" : 2,
  "lastExtentSize" : 10066176,
  "paddingFactor" : 1,
  "flags" : 1,
  "totalIndexSize" : 27983872,
  "indexSizes" : {
```

① 还可以键入范围更明确的show tables。

```
    "_id_" : 21307392,
    "num_1" : 6676480
 },
  "ok" : 1
}
```

结果文档中的一些值仅在复杂的调试情况中才会有用。但最起码能知道指定集合和它的索引到底占用了多少空间。

2.3.2 命令工作原理

与截至目前本章所描述的插入、更新、删除和查询操作不同，某些MongoDB操作是*数据库命令*。数据库命令一般都是管理类命令，比如之前提到的`stats()`方法，但它们也可能用于控制诸如MapReduce之类的核心MongoDB特性。

不管这些命令的功能是什么，它们的共同点是，在实现上它们都是对名为`$cmd`的特殊虚集合的查询。要明白这是什么意思，来看一个简单的例子。还记得我们是如何调用`stats()`数据库命令的吗：

```
> db.stats()
```

`stats()`是一个辅助方法，它封装了Shell的命令调用方法。可以输入下列等效操作：

```
> db.runCommand( {dbstats: 1} )
```

该操作的输出和`stats()`方法的输出是一样的。请注意，命令是由文档`{dbstats: 1}`来定义的。一般来说，我们可以向`runCommand`方法传递文档定义，借此运行各种命令。下面是运行集合统计命令的方法：

```
> db.runCommand( {collstats: 'numbers'} )
```

命令的输出你一定不会陌生。

要深入了解数据库命令，我们应该看看`runCommand()`方法到底是怎么工作的。这并不难知道，因为MongoDB Shell会输出所有方法的实现，只要这些方法忽略括号就行了。我们可以改变下面的命令：

```
> db.runCommand()
```

执行无括号的版本，一探究竟：

```
> db.runCommand
  function (obj) {
    if (typeof obj == "string") {
      var n = {};
      n[obj] = 1;
      obj = n;
    }
    return this.getCollection("$cmd").findOne(obj);
  }
```

函数中的最后一行无非就是查询$cmd集合。给它下个恰当的定义,数据库命令是对特殊集合$cmd的查询,查询选择器就是对命令本身的定义,仅此而已。你能想到如何手工运行集合统计命令吗?这很简单:

```
db.$cmd.findOne( {collstats: 'numbers'} );
```

使用runCommand辅助方法会更简单一点，但能了解内部细节总是好的。

2.4 获得帮助

目前为止，我们已经证实了MongoDB Shell的价值，可以用它试验数据和数据库管理操作。但是，既然我们可能会在Shell中花费大量的时间，最好能知道如何获得帮助。

内置的帮助命令应该是首先要考虑的。db.help()会列出操作数据库对象的常用方法，执行db.foo.help()会列出操作集合的常用方法。

Shell中还有内置的Tab补全机制，输入方法的前几个字母后按两下Tab键，你会看到所有匹配的方法。下面是对所有以get打头的集合方法的Tab补全：

```
> db.foo.get
db.foo.getCollection(    db.foo.getIndexSpecs(    db.foo.getName(
db.foo.getDB(            db.foo.getIndexes(       db.foo.getShardVersion(
db.foo.getFullName(      db.foo.getIndices(
db.foo.getIndexKeys(     db.foo.getMongo(
```

如果有更大的雄心壮志，又熟悉JavaScript，那么Shell能让你很轻松地查看任意指定方法的实现。举例来说，假设你想知道save()方法到底是如何工作的。你当然可以查阅MongoDB的源代码，但还有更简单的方法，只需键入不带执行括号的方法名即可。我们一般是这样执行save()的：

```
> db.numbers.save({num: 123123123});
```

这是查看save()实现的方法：

```
> db.numbers.save
function (obj) {
    if (obj == null || typeof obj == "undefined") {
        throw "can't save a null";
    }
    if (typeof obj._id == "undefined") {
        obj._id = new ObjectId;
        return this.insert(obj);
    } else {
        return this.update({_id:obj._id}, obj, true);
    }
}
```

仔细阅读函数定义，你会发现save()只是对insert()和update()的封装。如果正保存的对象没有_id字段，这个方法会添加该字段，调用insert()；否则执行更新。

这个查看Shell方法的小技巧很好用，在探索MongoDB Shell时记得要多用。

2.5　小结

你已经看过了实践中的文档数据模型，在此之上，我们也演示了很多不同的常用MongoDB操作。你已经了解了如何创建索引，并且通过`explain()`看到了基于索引提升性能的真实例子。此外，你应该可以获取系统中与集合和数据库相关的信息，也了解了`$cmd`集合的知识，如果遇到问题，还可以通过一些小技巧来寻求帮助。

在MongoDB Shell中可以学到很多东西，但没什么能代替构建真实应用程序的经验。所以下一章我们要从一个无忧无虑的数据游乐场切换到真实的数据车间。你将了解驱动是如何工作的，随后使用Ruby驱动构建一个简单的应用程序，用一些真实的数据来体验MongoDB。

第3章 使用MongoDB编写程序

本章内容

- ❑ 通过 Ruby 介绍 MongoDB API
- ❑ 驱动的工作原理
- ❑ BSON 格式与 MongoDB 网络协议
- ❑ 构建完整的示例应用程序

是时候行动起来了。虽然在MongoDB Shell的实验里还有很多东西要学，但只有在用它做过东西之后你才能发现它的真实价值，也就是要动手编程，并认识一下MongoDB驱动。正如之前提到过的，10gen为几乎所有流行编程语言都提供了有官方支持的、遵循Apache协议的MongoDB驱动。本书的驱动示例使用的是Ruby语言，但我所要描述的原理则放之四海而皆准，能很轻松地套用到其他驱动上。如果你求知欲旺盛，附录D中有PHP、Java和C++的驱动API。

初识Ruby？

Ruby是一门流行的、可读性很高的脚本语言。书中代码示例的设计非常浅显易懂，因此就算是不熟悉Ruby的开发者也能从中获益。如果有难以理解的Ruby惯用法出现，我会在文中做解释。如果你想花些时间了解一下Ruby，可以先从官方的20分钟教程（参见http://mng.bz/THR3）开始。

我们将分三个步骤来探究使用MongoDB的编程。首先，安装MongoDB Ruby驱动并介绍基本的CRUD操作。这一步会很快而且你会很熟悉，因为驱动API和Shell里用到的东西很类似。其次，我们会深入驱动之中，解释它是如何连接MongoDB的。这节的内容也不会过于深入底层，而是介绍一般情况下驱动背后做的事情。最后，我们将开发一个简单的Ruby应用程序，用它来监控Twitter。用了真实的数据集，我们会看到MongoDB在现实场景中是如何工作的。本章还为第二部分中更深层次的示例打下了基础。

3.1 通过 Ruby 使用 MongoDB

一般在想到驱动时，映入脑海的都是低级的位操作和迟钝的接口。感谢上帝，MongoDB的

语言驱动和这一点儿都不沾边，API反而设计得很直观、很对语言的胃口，因此很多应用程序索性直接把MongoDB驱动作为与数据库通信的唯一接口。驱动API在不同语言之间保持着相当的一致性，这意味着，如果需要，开发者可以轻松地在语言之间进行切换。如果你是一位应用程序开发者，会发现在使用任何MongoDB驱动时都感觉良好且生产率很高，不用自己操心底层的实现细节。

本节将带你安装MongoDB Ruby驱动，连接数据库，了解如何执行基本的CRUD操作。这将为本章最后要构建的应用程序打下基础。

3.1.1 安装与连接

我们可以使用RubyGems安装MongoDB Ruby驱动，RubyGems是Ruby的包管理系统。

注意 如果还没在系统上安装Ruby，可以在http://www.ruby-lang.org/en/downloads/ 找到详细的安装指南。你还需要Ruby的包管理器RubyGems，可以在http://docs.rubygems.org/read/chapter/3 找到RubyGems的安装指南。

```
gem install mongo
```

这条命令会安装mongo和bson[①] Gem。我们应该会看到如下输出（版本号可能会比下面的更高）：

```
Successfully installed bson-1.4.0
Successfully installed mongo-1.4.0
2 gems installed
Installing ri documentation for bson-1.4.0...
Installing ri documentation for mongo-1.4.0...
Installing RDoc documentation for bson-1.4.0...
Installing RDoc documentation for mongo-1.4.0...
```

我们从连接MongoDB开始。首先确保mongod正在运行，接下来创建一个名为connect.rb的文件，键入以下代码：

```
require 'rubygems'
require 'mongo'

@con   = Mongo::Connection.new
@db    = @con['tutorial']
@users = @db['users']
```

头两条require语句保证一定加载了驱动，接下来的三行实例化了一个连接，将tutorial数据库分配给了@db变量，在@users变量中保存了一个对users集合的引用。保存并运行文件：

```
$ ruby connect.rb
```

---

① BSON会在下一节中做详细说明，它是一种受JSON启发的二进制格式，MongoDB用它来表示文档。bson Ruby Gem能将Ruby对象序列化为BSON，反之亦然。

如果没有抛出异常，那么你已经成功地用Ruby连接到MongoDB了。虽然不够诱人，但连接是任何语言使用MongoDB的第一步。接下来，我们使用该连接插入文档。

3.1.2　用Ruby插入文档

所有MongoDB驱动在设计上都要求使用其语言中最自然的文档表述方式。在JavaScript中JSON对象是最明显的选择，因为JSON是一种文档数据结构；在Ruby中，散列数据结构最为合适。原生的Ruby散列和JSON对象只是稍有不同，最明显的是JSON用冒号来分隔键和值，而Ruby则使用=>[①]。

如果你是一路跟着示例做下来的，那么继续向connect.rb文件添加代码。你也可以选择另一种不错的方式，即使用Ruby的交互式REPL——irb。你可以运行irb，载入connect.rb，这样立刻就能访问到其中实例化的连接、数据库和集合对象了。接着可以运行Ruby代码并接收实时反馈。下面就是一个例子：

```
$ irb -r./connect.rb
irb(main):001:0> id = @users.save({"lastname" => "knuth"})
=> BSON::ObjectId('4c2cfea0238d3b915a000004')
irb(main):002:0> @users.find_one({"_id" => id})
=> {"_id"=>BSON::ObjectId('4c2cfea0238d3b915a000004'), "lastname"=>"knuth"}
```

让我们为users集合构建一些文档。创建两个文档来表示用户smith和jones。每个文档都用Ruby散列来表示并被分配一个变量：

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

要保存文档，将它们传给集合的insert方法即可。每次调用insert都会返回一个唯一ID，应该将它保存在变量里以便日后获取数据：

```
smith_id = @users.insert(smith)
jones_id = @users.insert(jones)
```

可以通过一些简单的查询来验证文档是否成功保存。通常每个文档的对象ID都会被保存在_id键中。可以通过用户集合的find_one方法来进行查询：

```
@users.find_one({"_id" => smith_id})
@users.find_one({"_id" => jones_id})
```

如果你是在irb里运行代码的，查询的返回值会显示在提示符中。如果是运行Ruby文件，加上Ruby的p方法，把结果输出到屏幕上：

```
p @users.find_one({"_id" => smith_id})
```

你已经成功地用Ruby插入了两个文档，现在再来仔细看看查询。

3.1.3　查询与游标

你刚使用了驱动的find_one方法来获取单条结果。能这么简单是因为find_one隐藏了一些

① 在Ruby 1.9中，也可以将冒号作为键值分隔符，但为了保证向后兼容性，本书中仅使用=>。

MongoDB查询的细节。通过标准的find方法能对此有所了解，以下是两个可能的对数据集的查找操作：

```
@users.find({"last_name" => "smith"})

@users.find({"age" => {"$gt" => 20}})
```

很明显，第一个查询找出了所有last_name是smith的用户文档，第二个查询匹配所有age大于20的文档。试着在irb中键入第二个查询：

```
irb(main):008:0> @users.find({"age" => {"$gt" => 30}})
=> <#Mongo::Cursor:0x10109e118 ns="tutorial.users"
      @selector={"age" => "$gt" => 30}}>
```


你将发现的第一件事会是find方法并不返回结果集，而是一个游标对象。游标出现在很多数据库系统中，出于对效率的考虑，迭代地批量返回查询结果集。假设users集合包含100万个匹配查询的文档。如果没有游标，就必须一次性返回全部这些文档。立刻返回这么大的结果意味着将所有数据复制到内存里，通过网络进行传输，然后反序列化到客户端。这本不应是个资源密集型操作，为了防止这种情况，查询实例化了一个游标，以一个可控的分块大小来获取结果集。当然，这对用户而言是透明的；在按需通过游标请求更多结果、连续调用MongoDB时，会填充驱动的游标缓冲。

下一节中会更详细地解释游标。回到例子上，现在获取到$gt查询的结果：

```
cursor = @users.find({"age" => {"$gt" => 20}})

cursor.each do |doc|
  puts doc["last_name"]
end
```

这里用到了Ruby的each迭代器，它将每个结果都传递给一个代码块，本例中，稍后会将last_name属性输出到控制台。如果你不熟悉Ruby的迭代器，下面是一段更语言中立的等效代码：

```
cursor = @users.find({"age" => {"$gt" => 20}})

while doc = cursor.next
  puts doc["last_name"]
end
```

这个例子里，我们连续调用游标的next方法，将值赋给本地变量doc，使用一个简单的while循环对游标进行迭代。

回想上一章里的Shell示例，再想想本节的游标，你会感到大吃一惊。Shell中使用游标的方式与驱动一样，不同之处在于调用find()时Shell会自动迭代前20个游标结果。要获取剩下的结果，可以通过it命令继续手工迭代。

3.1.4 更新与删除

注意，上一章里的更新操作要求至少有两个参数：一个查询选择器和一个更新文档。下面是一个使用Ruby驱动的简单示例：

```
@users.update({"last_name" => "smith"}, {"$set" => {"city" => "Chicago"}})
```

这个更新先查找last_name是smith的第一个用户，如果找到的话就将它的city值设置为Chicago，其中使用了$set操作符。

默认情况下，MongoDB只会更新单个文档。就算你有多个用户的姓是smith，也只会更新一个文档。要将更新应用到特定的smith上，需要向查询选择器添加更多的条件。但如果是想更新所有的smith文档，必须发起*多项更新*（multi-update）。为此，我们可以将:multi => true作为第三个参数传递给update方法：

```
@users.update({"last_name" => "smith"},
  {"$set" => {"city" => "New York"}}, :multi => true)
```

删除数据更加简单，使用remove方法就可以了。该方法接受一个可选的查询选择器，只删除那些匹配选择器的文档。如果没有提供选择器，就删除集合中的所有文档。此处，我们要删除age属性值大于等于40的所有用户文档：

```
@users.remove({"age" => {"$gte" => 40}})
```

如果不带参数，remove方法会删除所有的文档：

```
@users.remove
```

在上一章里我们说过remove实际上并不会删除集合，要删除集合及其索引，可以使用drop_collection方法：

```
connection = Mongo::Connection.new
db = connection['tutorial']
db.drop_collection('users')
```

3.1.5　数据库命令

在上一章里我们已经见到过数据库命令了，并看了两个stats命令。此处，我们将了解如何在驱动中运行命令，例子就是listDatabases命令，这是几个必须在admin数据库上运行的命令之一，在开启身份验证的时候还做了特殊处理。关于身份验证与admin数据库的详细内容，请阅读第10章。

首先，实例化一个Ruby数据库对象指向admin数据库。然后将命令的*查询说明*（query specification）传给command命令：

```
@admin_db = @con['admin']
@admin_db.command({"listDatabases" => 1})
```

执行的响应是一个Ruby散列，罗列了所有存在的数据库和其在磁盘上的大小：

```
{
    "databases" => [
        {
            "name" => "tutorial",
            "sizeOnDisk" => 218103808,
            "empty" => false
        },
```

```
        {
            "name" => "admin",
            "sizeOnDisk" => 1,
            "empty" => true
        },
        {
            "name" => "local",
            "sizeOnDisk" => 1,
            "empty" => true
        }
    ],
    "totalSize" => 218103808,
    "ok" => true
}
```


一旦习惯了使用Ruby散列来表示文档，几乎就可以无缝地从Shell API过渡过来。如果你还是对通过Ruby使用MongoDB感到不安，请不用担心，3.3节将带你进行更多的练习。但现在我们要稍作停顿，了解一下MongoDB驱动是如何工作的，这能让人更多地了解MongoDB的设计，以便能更有效地使用驱动。

3.2 驱动是如何工作的

现在你一定会对通过驱动或MongoDB Shell发出命令后究竟发生了什么感到好奇。本节中，我们会掀开“帘子”看看驱动是如何序列化数据并将它传给数据库的。

所有的MongoDB驱动都有三个主要功能。首先，生成MongoDB对象ID，这是存储在所有文档_id字段里的默认值。其次，驱动会把所有语言特定的文档表述和BSON互相转换，BSON是MongoDB使用的二进制数据格式。前面的例子中，驱动将所有Ruby散列都序列化成了BSON，然后再把数据库返回的BSON反序列化成Ruby散列。

最后一个功能是使用MongoDB的网络协议通过TCP套接字与数据库通信。协议的具体内容超出了我们的讨论范围。但套接字通信的风格很重要，尤其是通过套接字写入时是否要等待响应，本节中我们会探讨这个话题。

3.2.1 对象ID生成

每个MongoDB文档都要求有一个主键，它在每个集合中对于所有文档必须是唯一的，主键存放在文档的_id字段中。开发者可以随意使用自定义值作为_id，但如果没有提供该值，就会使用MongoDB对象ID。在向服务器发送文档前，驱动会检查是否提供了_id字段，如果没有则生成一个适当的对象ID，存储为_id。

因为MongoDB对象ID是全局唯一的标识符，所以可以安全地在客户端为文档分配ID，不用担心会有重复ID。现在，你已经看到过真实的对象ID了，但可能没有注意到它们是由12个字节构成的。如图3-1所示，这些字节是有特定结构的。



图3-1　MongoDB对象ID格式

最开头的4字节是标准的Unix时间戳，编码了从新纪元开始的秒数。接下来的3字节存储了机器ID，随后则是2字节的进程ID。最后3字节存储了进程局部的计数器，每次生成对象ID计数器都会加1。

使用MongoDB对象ID带来的好处之一是其中包含了时间戳。大多数驱动都允许方便地提取时间戳，从而提供文档的创建时间，精度是最接近的一秒钟。使用Ruby驱动，可以调用对象ID的`generation_time`方法来获得ID的创建时间，返回值是Ruby的`Time`对象：

```
irb(main):002:0> id = BSON::ObjectId.new
=> BSON::ObjectId('4c41e78f238d3b9090000001')
irb(main):003:0> id.generation_time
=> Sat Jul 17 17:25:35 UTC 2010
```

很自然的，我们还可以使用对象ID根据对象的创建时间进行范围查询。举个例子，如果希望查询所有在2010年10月至2010年11月之间创建的文档，可以创建两个对象ID，将它们的时间戳分别编码为那两个时间，然后对`_id`发起范围查询。Ruby提供了从任意`Time`对象创建对象ID的方法，因此实现这一功能的代码很简单：

```
oct_id = BSON::ObjectId.from_time(Time.utc(2010, 10, 1))
nov_id = BSON::ObjectId.from_time(Time.utc(2010, 11, 1))

@users.find({'_id' => {'$gte' => oct_id, '$lt' => nov_id}})
```

我已经解释了MongoDB对象ID的基本原理及各个字节背后的含义。剩下的就是了解它们是如何编码的，这是下一节的主题，届时我们还将讨论BSON。

3.2.2　BSON

BSON是MongoDB中用来表示文档的二进制格式，它既是存储格式，也是命令格式：所有文档都以BSON格式存储在磁盘上，所有查询和命令都用BSON文档来指定。因此，所有的MongoDB驱动必须能在语言特定的文档表述和BSON之间进行转换。

BSON定义了能在MongoDB中使用的数据类型。知道BSON包含哪些类型，了解它们的编码，这对有效使用MongoDB以及发生性能问题时的诊断都大有好处。

在本书编写时，BSON规范中包含了19种数据类型。这就是说，文档中的每个值为了能存储在MongoDB里，必须要能转换为这19种类型中的一种。BSON类型包含了很多我们所期待的类型：UTF-8字符串、32位和64位整数、双精度浮点数、布尔值、时间戳和UTC日期时间（datetime）。但是，还有一部分类型是特定于MongoDB的。举例来说，上一节中描述的对象ID格式就有自己的类型；有针对模糊大字段（opaque blob）的二进制类型；如果语言支持的话，MongoDB里甚至还提供了符号类型（symbol type）。

图3-2描述了如何将一个Ruby散列序列化为正确的BSON文档。Ruby文档中包含一个对象ID和一个字符串。在转换为BSON文档后，头部的4字节表明了文档的大小（可以看到此处是38字节）。接下来是两个键值对，每对都由一个表示其类型的字节开头，随后是由null结尾的字符串表示键名，然后是被存储的值，最后是一个null字节表示文档结束。

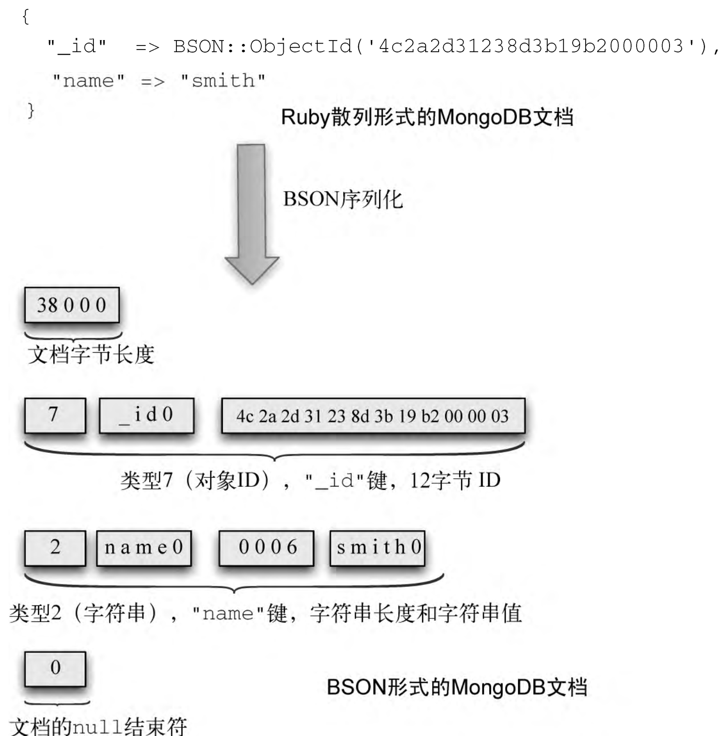


图3-2 从Ruby转换为BSON

虽然不一定要知道BSON的详情，但经验表明了解BSON对MongoDB开发者是有好处的。举个例子，将对象ID表示成字符串或者BSON对象ID这两种做法都是正确的。因此，以下两个Shell查询并不等价：

```
db.users.find({_id : BSON::ObjectId('4c41e78f238d3b9090000001')});
db.users.find({_id : '4c41e78f238d3b9090000001'})
```

其中只有一个查询能匹配_id字段，这完全取决于users集合中的文档存储的是BSON对象ID，还是表示ID十六进制值的BSON字符串。[①]这个例子说明即使只对BSON略知一二，在诊断简单代码问题时都很有帮助。

① 顺便说一下，如果要保存MongoDB对象ID，应该使用BSON对象ID，而不是字符串。除了遵循对象ID的存储惯例，BSON对象ID还能比字符串节省一半以上的空间。

3.2.3　网络传输

除了创建对象ID以及序列化到BSON，MongoDB驱动还有一项核心功能：与数据库服务器通信。如前文所述，通信是基于TCP套接字的，使用了自定义网络协议。[①]这个TCP的工作是相当底层的，大多数应用程序开发者对此也并不关心。此处与开发者相关的是要理解驱动何时会等待服务器的响应，何时又能不必等待响应。

我已经解释过查询是如何工作的，很显然，查询必须要有一个响应。回顾一下，当游标对象的`next`方法被调用后即会发起一次查询。这时会把查询发给服务器，其响应是一批文档。如果这批文档能满足查询，则不必再和服务器进行通信。但如果查询结果较多，恰好无法全部放进第一个服务器响应中，将会向服务器发送一个所谓的`getmore`指令获取下一批查询结果。随着游标的迭代，在查询结束前会连续不断地调用`getmore`方法。

上述查询的网络行为并没有什么好让人惊讶的，但说到数据库写操作（插入、更新及删除），默认的行为看起来就不怎么正统了。这是因为在向服务器写数据时，驱动默认不会等待服务器的响应。因此在插入文档时，驱动会向套接字写数据并假设写入是成功的。让这种做法能成为现实的一种策略就是客户端生成对象ID：既然已经有了文档的主键，就没有必要等待服务器返回该主键了。

这种不关心结果的写策略让很多用户如坐针毡；幸运的是，该行为是可配置的。所有的驱动都实现了一个安全写入模式，对所有的写操作（插入、更新及删除）都能开启该模式。在Ruby中，能像这样发起一次安全插入：

```
@users.insert({"last_name" => "james"}, :safe => true)
```

以安全模式写入时，驱动会在插入消息后追加一条特殊的`getlasterror`命令。它将做两件事。第一，`getlasterror`是一条命令，因此需要和服务器做一次通信，这保证了写操作已经送达服务器。第二，该命令验证了服务器在当前连接中没有抛出任何错误。如果有错误被抛出，驱动会发出一个异常，这一异常能被优雅地处理。我们可以使用安全模式来保证应用程序的关键写操作到达服务器，也可以在期待显式错误时使用安全模式。举例来说，经常要强调值的唯一性。如果正在保存用户数据，我们会维护一个`username`字段的唯一性索引。在有重复`username`时，该唯一性索引会造成文档插入失败，但要知道插入失败的唯一途径就是使用安全模式。

大多数情况中，慎重的做法是默认开启安全模式。随后，针对一些写入少但要求高吞吐量的应用程序部分可以选择关闭安全模式。要做这种权衡并不容易，还有更多安全模式选项要考虑。在第8章中我们将就此进行更详细的讨论。

目前为止，你了解了驱动是如何工作的，应该感到更舒服了，也许还迫不及待地想要构建一个真实的应用程序。在下一节里，我们会结合所有的知识，使用Ruby驱动来构建一个基本的Twitter监控应用。

---

① 一些驱动还支持Unix 域套接字通信。

3.3　构建简单的应用程序

我们将构建一个简单的应用程序，用来归档及显示微推文。我们可以把它想象成更大的应用程序中的一个组件，这个应用允许用户密切注意与其业务相关的搜索项。该示例将展示处理来自Twitter API之类数据源的JSON，以及将它转成MongoDB文档有多容易。如果使用关系型数据库，就不得不事先设计一个Schema，可能还会包含多张数据表，然后还要声明那些表。使用MongoDB的话，这些事情就都不需要了，但还能保留推文文档丰富的结构，并且可以高效地进行查询。

我们称该应用为TweetArchiver，它由两个组件组成：归档器和查看器，归档器会调用Twitter的搜索API保存相关推文，查看器用于在Web浏览器里浏览结果。

3.3.1　配置

该应用程序会用到三个Ruby库，可以这样进行安装：

```
gem install mongo
gem install twitter
gem install sinatra
```

有个配置文件能在归档器和查看器脚本之间进行共享会很有用，创建一个名为config.rb的文件，初始化如下常量：

```
DATABASE_NAME   = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["mongodb", "ruby"]
```

首先指定了应用程序中使用的数据库和集合的名字。然后定义了一个搜索项数组，我们会把它们发给Twitter API。

接下来是编写归档器脚本。先从`TweetArchiver`类开始，用一个搜索项来进行实例化。然后调用`TweetArchiver`实例的`update`方法，这会发起一次Twitter API调用，将结果保存到MongoDB集合里。

让我们先从类的构造器下手：

```
def initialize(tag)
  connection = Mongo::Connection.new
  db         = connection[DATABASE_NAME]
  @tweets    = db[COLLECTION_NAME]

  @tweets.create_index([['id', 1]], :unique => true)
  @tweets.create_index([['tags', 1], ['id', -1]])

  @tag = tag
  @tweets_found = 0
end
```

`initialize`方法实例化了一个连接、一个数据库对象和用来存储推文的集合对象，其中还创建了两个索引。每条推文都有一个`id`字段（与MongoDB的`_id`字段不同），代表推文的内部Twitter ID。我们为这个字段创建了一个唯一性索引，以避免同一条推文被插入两次。

我们还在`tags`和`id`字段上创建了一个组合索引，`tags`升序，`id`降序。索引可以指定是升序

还是降序，这主要在创建组合索引时比较重要，应该总是基于自己期待的查询模式来选择方向。因为我们希望查询特定的标签，并且按时间由近及远显示结果，所以tags升序、id降序的索引既能用来过滤结果，也能用来进行排序。如你所见，可以用1表示升序、-1表示降序，以此来指明索引方向。

3.3.2　收集数据

在MongoDB中可以插入数据而无需考虑其结构。因为不用事先知道会有哪些字段，Twitter可以随意修改API的返回值，不会给应用程序带来什么不良后果。一般来说，如果使用RDBMS，对Twitter API（说得更广泛点，对数据源）的任何改动都会要求进行数据库Schema迁移。用了MongoDB，应用程序可能需要做些修改来适应新的数据Schema，但数据库本身可以自动处理各种文档风格的Schema。

Ruby的Twitter库返回的是Ruby散列，因此可以直接将其传递给MongoDB集合对象。在TweetArchiver中，添加如下实例方法：

```
def save_tweets_for(term)
  Twitter.search(term).each do |tweet|
    @tweets_found  += 1
    tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
    @tweets.save(tweet_with_tag)
  end
end
```

在保存每个推文文档前，要做个小修改。为了简化日后的查询，将搜索项添加到tags属性中。然后将修改过的文档传递给save方法。代码清单3-1中是完整的归档器代码。

代码清单3-1　抓取推文并将其归档在MongoDB中的类

```
require 'rubygems'
require 'mongo'
require 'twitter'

require File.join(File.dirname(__FILE__), 'config')

class TweetArchiver

  # Create a new instance of TweetArchiver
  def initialize(tag)
    connection = Mongo::Connection.new
    db         = connection[DATABASE_NAME]
    @tweets    = db[COLLECTION_NAME]
    @tweets.create_index([['id', 1]], :unique => true)
    @tweets.create_index([['tags', 1], ['id', -1]])

    @tag = tag
    @tweets_found = 0
  end

  def update
    puts "Starting Twitter search for '#{@tag}'..."
    save_tweets_for(@tag)
    print "#{@tweets_found} tweets saved.\n\n"
```

```
  end

  private

  def save_tweets_for(term)
    Twitter.search(term).each do |tweet|
      @tweets_found += 1
      tweet_with_tag = tweet.to_hash.merge!({"tags" => [term]})
      @tweets.save(tweet_with_tag)
    end
  end

end
```


剩下的就是要编写一个脚本，为每个搜索项运行`TweetArchiver`代码。创建update.rb，包含以下代码：

```
require File.join(File.dirname(__FILE__), 'config' )
require File.join(File.dirname(__FILE__), 'archirer' )

TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
end
```

然后，运行该更新脚本：

```
$ ruby update.rb
```

我们会看到一些状态消息，它们指明程序找到并保存了推文。可以打开MongoDB Shell，直接查询集合来验证脚本是否能正常运行：

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

为了保证归档内容始终是最新的，可以使用一个cron任务，每隔几分钟就运行一次更新脚本。但那是管理的细节，这里的重点是通过寥寥几行代码就能保存从Twitter查到的推文。[①]接下来的任务是显示结果。

3.3.3 查看归档

我们将使用Ruby的Sinatra Web框架构建一个简单的应用，用来显示结果。创建一个名为viewer.rb的文件，和其他脚本放在同一目录里。随后，新建views子目录，放入一个名为tweets.erb的文件。项目结构看起来应该像下面这样：

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb
```

① 还可以用更少的代码来实现这一功能，这就留给读者作为练习了。

现在编辑viewer.rb，加入以下代码。

代码清单3-2　一个简单的Sinatra应用程序，用于显示并搜索Tweet归档

```
require 'rubygems'
require 'mongo'                                  ❶ 加载库
require 'sinatra'                                ◁

require File.join(File.dirname(__FILE__), 'config')

configure do                                              ❷ 实例化tweets集合
  db = Mongo::Connection.new[DATABASE_NAME]
  TWEETS  = db[COLLECTION_NAME]                          ◁
end

get '/' do
  if params['tag']                                                ❸ 动态构建查询选择器
    selector = {:tags => params['tag']}                          ◁
  else                                             ❹ 或者使用空白选择器
    selector = {}                                 ◁
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])       ◁—❺ 发起查询

  erb :tweets                                  ◁—❻ 呈现视图
end
```

前面几行代码加载了所需的库，还有配置文件❶。接下来的配置块中创建了一个到MongoDB的连接，并把指向`tweets`集合的引用保存在常量`TWEETS`里❷。

应用程序中最重要的部分是`get '/' do`之后的代码，这个块里的代码处理了对应用程序根URL的请求。首先，构建查询选择器：如果提供了URL参数`tags`则创建一个查询选择器，将结果集限定在给定标签里❸；否则就创建一个空白的选择器，查询会返回集合中的全部文档❹。然后发起查询❺。现在你应该知道赋给`@tweets`变量的不是结果集，而是一个游标，我们将在视图中对该游标进行迭代。

最后一行❻呈现了视图文件tweets.erb，完整代码如代码清单3-3所示。

代码清单3-3　用于显示推文的内嵌Ruby的HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang='en' xml:lang='en' xmlns='http://www.w3.org/1999/xhtml'>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

  <style>
    body {
      background-color: #DBD4C2;
      width: 1000px;
      margin: 50px auto;
    }

    h2 {
      margin-top: 2em;
```

```
    }
  </style>

</head>

<body>

<h1>Tweet Archive</h1>

<% TAGS.each do |tag| %>
  <a href="/?tag=<%= tag %>"><%= tag %></a>
<% end %>

<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['from_user'] %>">
      <%= tweet['from_user'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>

  <img src="<%= tweet['profile_image_url'] %>" width="48" />
<% end %>

</body>
</html>
```

3

大部分代码只是混入了ERB的HTML，[①]其中的重要部分在结尾附近，有两个迭代器。第一个迭代器遍历了标签列表，显示的链接能将结果集限定在指定的标签上。`@tweets.each`开头的是第二个迭代器，遍历了每条推文，显示推文的正文、创建日期和用户头像图片。运行应用程序来查看结果：

```
$ ruby viewer.rb
```

如果应用程序正常启动，我们将看到标准的Sinatra启动消息：

```
$ ruby viewer.rb
== Sinatra/1.0.0 has taken the stage on 4567 for development
with backup from Mongrel
```

我们可以打开Web浏览器，访问http://localhost:4567，页面应该会和图3-3类似。单击屏幕上方的链接可以缩小结果范围，基于特定的标签显示结果。

应用程序就这样完成了，不可否认它比较简单，但它演示了MongoDB的易用性。我们不用事先定义Schema；能充分利用二级索引加速查询，避免重复插入；还能相对简单地和编程语言进行集成。

① ERB全称是embedded Ruby。Sinatra应用通过一个ERB处理器来运行tweets.erb文件，并在应用程序上下文中运算`<%`和`%>`之间的Ruby代码。

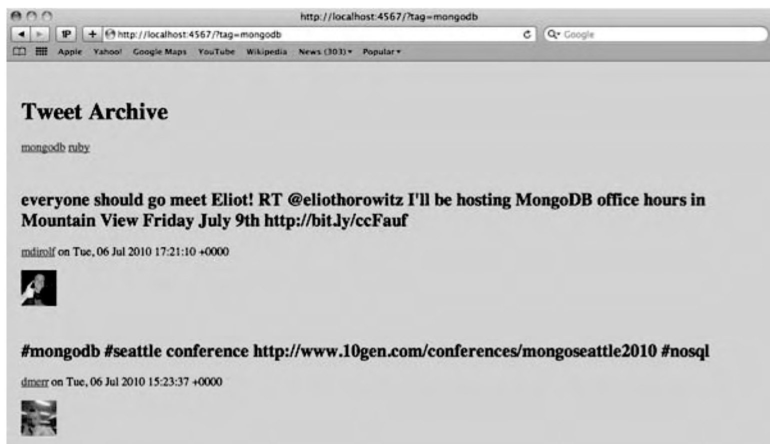


图3-3　Web浏览器中呈现的推文归档

3.4　小结

我们刚刚学习了通过Ruby编程语言同MongoDB交互的基础知识，看到了使用Ruby表示文档有多简单，Ruby的CRUD API和MongoDB Shell里的CRUD有多相似。我们深入其中，大致了解了驱动是如何构建的以及对象ID、BSON和MongoDB网络协议的细节。最后，我们还构建了一个简单的应用程序，结合真实数据来演示MongoDB的使用。虽然我们还不能自称MongoDB大师，但应该已经能用它编写应用程序了。

从第4章开始，我们将详细讨论到目前为止学到的东西，尤其会探讨如何使用MongoDB来构建电子商务应用。那将是个庞大的项目，而我们只关注后端的一些部分。我会展示该领域中的一些数据模型，你将了解到如何对那些数据做插入和查询。

Part 2

第二部分

MongoDB 与应用程序开发

本书的第二部分会深入剖析MongoDB的文档数据模型、查询语言和CRUD操作（创建、读取、更新和删除）。

我们会渐进地设计一个电子商务数据模型，以及管理这些数据所必需的 CRUD 操作，在此过程中具体讨论上述几个话题。因此，每章都会以自顶向下的方式展现其主题内容，先给出示例电子商务应用程序领域里的例子，然后系统地描述各个细节。一开始，你可能只想了解电子商务示例，然后再了解细节内容，反之亦然。

在第 4 章里，你将了解到一些 Schema 设计原则，随后为产品、分类、用户、订单和产品评论构造基本的电子商务数据模型。你还将了解到 MongoDB 如何组织数据库、集合和文档级别的数据。其中还会包含一个 BSON 核心数据类型的小结。

第 5 章涉及了 MongoDB 的查询语言和聚合函数。你将了解到如何对上一章里开发的数据模型发起常用查询，还会练习使用一些聚合函数。在“具体细节”部分，你会看到查询操作符详细的语义。本章结尾处解释了 MapReduce 和分组函数。

第 6 章通过 MongoDB 的更新和删除操作，为我们完整呈现了电子商务数据模型的理论依据。你将了解到如何维护分类层级，如何事务性地管理库存。此外，这一章还会详细介绍更新操作符，涉及强大的 `findAndModify` 命令。

第 4 章

面向文档的数据

4

本章内容

- Schema 设计
- 电子商务数据模型
- 数据库、集合与文档

本章详细介绍了面向文档的数据建模，以及数据库、集合与文档级别的数据在MongoDB中是如何组织的。我会先简单阐述一下Schema的设计，这是很有帮助的，因为大量MongoDB用户从未给传统RDBMS以外的数据库设计过Schema。此处讨论到的原则为本章第二部分做了铺垫，第二部分里我们会看到一个MongoDB的电子商务Schema。通篇你会看到这个Schema与等价的RDBMS的Schema有何区别，还会了解到MongoDB中典型的实体关系是如何表示的，比如一对多和多对多的实体关系。该电子商务Schema还会作为后续各章中讨论查询、聚合与更新的基础。

既然文档是MongoDB原生的东西，我将用本章的最后部分来讨论文档及其周边的小细节与边边角角的内容。这意味着相比你目前所掌握的知识，本章会更详细地讨论数据库、集合与文档。如果能读到最后，你就会熟悉MongoDB文档数据最晦涩的特性与局限。以后也许你还会来阅读本章的最后一节，因为其中包含了很多在实际使用MongoDB的过程中会遇到的陷阱。

4.1 Schema 设计原则

设计数据库Schema是在已知数据库系统特性、数据本质以及应用程序需求的情况下为数据集选择最佳表述的过程。关系型数据库系统的Schema设计原则已经很完整了，在RDBMS中鼓励使用正规化的数据模型，这能帮助确保可查询性，避免对数据的更新造成数据不一致。而且，已有的这些模式能避免开发者产生疑问，比如如何建模一对多和多对多关系。但就算是在关系型数据库中，Schema设计也不是一门精确的科学。高性能要求的应用程序或者需要处理非结构化数据的应用程序可能会要求一个更通用的数据模型。一些应用程序对存储和伸缩性要求颇高，以至于要打破所有旧的Schema设计规则。FriendFeed就是一个很好的例子，这里有篇描述该站点非传统数据模型的文章值得一读，详见http://mng.bz/ycG3。

如果你来自RDBMS的世界，MongoDB的这种缺乏硬性Schema设计规则的做法可能会让你感

到不太适应。虽然涌现出了一些好的实践，但对给定数据集的建模方法往往不止一个。本节的前提是其中介绍的原则能驱动Schema的设计，但现实情况是，那些原则都是可以变通的。在任何数据库系统中建模数据时，下面这些问题都值得考虑。

- *数据的基本单元是什么？* 在RDBMS中有带列和行的数据表。在键值存储中有指向不定类型值的键。在MongoDB中，数据的基本单元是BSON文档。
- *如何查询并更新数据？* 一旦理解了基本数据类型，我们需要知道如何操作数据。RDBMS有即时查询和联结操作查询。MongoDB也有即时查询，但不支持联结操作。简单的键值存储只能根据单个键来获取值。

  根据允许的更新类型，数据库也有所区分。RDBMS中，可以使用SQL以复杂的方式来更新文档，将多条更新封装在一个事务中以获得原子性，还能回滚。MongoDB不支持事务，但它支持多种原子更新操作，这些操作可作用于复杂文档的内部结构。简单键值存储中，可以更新一个值，但通常每次更新都是将值完全替换掉。

  其要点是构建最佳数据模型意味着理解数据库的特性。如果想在MongoDB里很好地建模数据，必须先理解它擅长于哪种查询和更新。
- *应用程序的访问模式是什么？* 除了理解数据的基本单元和数据库的特性，还需要明确应用程序的需求。如果你读了刚才提到的FriendFeed的文章，就会明白应用程序的特质能轻而易举地让Schema打破固有的数据建模原则。结论就是在确定理想的数据模型前，必须问无数个与应用程序有关的问题。读写比是多少？需要何种查询？数据是如何更新的？能想到什么并发问题？数据的结构化程度如何？

最好的Schema设计总是源于对正在使用的数据库的深入理解、对应用程序需求的准确判断以及过去的经验。本章的示例以及附录B中的Schema设计方式都将帮助你培养一种直觉，以便能出色地完成MongoDB中的Schema设计。学习了这些例子之后，你就能为自己的应用程序设计优秀的Schema了。

4.2　设计电子商务数据模型

下一代数据存储的演示一般都是围绕着社交媒体：以Twitter类演示应用居多。遗憾的是此类应用倾向于使用简单的数据模型。这就是为什么本章以及后续各章中要使用更丰富的电子商务领域模型了，其中包含了很多为人熟知的数据建模模式。而且也不难想象产品、分类、产品评论与订单是如何在RDBMS中建模的。这会让即将登场的示例更具启发性，因为可以将它们与预想的Schema设计进行对比。

电子商务通常是专属于RDBMS的一块领域，这是有原因的。首先，电子商务站点通常要求有事务，而事务是RDBMS的主要特性。其次，直到最近为止，要求有富数据模型和完善的查询的领域都会假定自己最适合RDBMS。下面的例子会对第二个假设提出质疑。

在继续之前，需要做一点说明。在本书中介绍如何构建完整的电子商务后端并不实际。我们要做的是选取少量的电子商务实体，演示如何在MongoDB中对其进行建模，尤其会关注产品与

分类、用户与订单，还有产品评论。针对每个实体，我都将展示示例文档。随后，我们还会看到一些数据库特性，它们能进一步补充文档的结构。

对很多开发者而言，数据建模总会伴随着对象映射，为此你可能使用过对象关系映射库，比如Java的Hibernate或者Ruby的ActiveRecord，这些库几乎就是在RDBMS上有效构建应用程序的必需品。但是MongoDB对此几乎没什么需要，部分原因是文档已经是类似对象的表述了。此外还和驱动有关，驱动为MongoDB提供了相当高阶的接口，仅用驱动接口就能在MongoDB之上构建完整的应用程序。

有人说，对象映射器很方便，因为它们有助于进行验证、类型检查和关联。很多成熟的MongoDB对象映射器在基本语言驱动之上又提供了一层额外的抽象，在大项目中可以考虑选择其一。[①]但是，不管是否使用对象映射器，最终总是在和文档打交道。这就是本章关注于文档本身的原因。知道在一个精心设计的MongoDB Schema里文档是什么样的，这能让你更好地使用该数据库，有没有对象映射器都是如此。

4.2.1　产品与分类

产品和分类是任何电子商务站点都必不可少的内容。在一个正规化的RDBMS模型中，产品倾向于使用大量的数据表，总会有张表用来存储基本产品信息，比如名称和SKU[②]，还有一些其他表用来关联送货信息和价格历史。如果系统允许产品带有任意属性，那么还需要一系列复杂的表来定义并存储那些属性，正如你在第1章的Magento示例中看到的那样。这种多表Schema在RDBMS表联结能力的帮助下很有用。

在MongoDB中对产品建模应该会简单很多，因为集合并不一定要有Schema，任何产品文档都可以容纳产品所需的各种动态属性。通过使用数组来容纳内部文档结构，还可以将RDBMS里的多表表述精简成一个MongoDB的集合。更具体一点，下面是一个取自园艺商店的示例产品。

代码清单4-1　示例产品文档

```
doc =
{ _id: new ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  name: "Extra Large Wheel Barrow",
  description: "Heavy duty wheel barrow...",

  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },
```

① 想知道哪个对象映射器是你语言里最流行的，可以看看http://mongodb.org里的建议。

② SKU是Stock Keeping Unit的缩写，即最小存货单位。——译者注

```
  total_reviews: 4,
  average_review: 4.5,
  pricing: {
    retail: 589700,
    sale: 489700,
  },

  price_history: [
      {retail: 529700,
       sale: 429700,
       start: new Date(2010, 4, 1),
       end: new Date(2010, 4, 8)
      },

      {retail: 529700,
       sale: 529700,
       start: new Date(2010, 4, 9),
       end: new Date(2010, 4, 16)
      },
  ],

  category_ids: [new ObjectId("6a5b1476238d3b4dd5000048"),
                 new ObjectId("6a5b1476238d3b4dd5000049")],

  main_cat_id: new ObjectId("6a5b1476238d3b4dd5000048"),

  tags: ["tools", "gardening", "soil"],

}
```

该文档包含基本的name、sku和description字段。_id字段里还存储着标准的MongoDB对象ID。此外，这里定义了一个短名称wheel-barrow-9092，以便提供有意义的URL。MongoDB的用户有时会抱怨URL里的对象ID太难看了，通常来说，你不会喜欢下面这样的URL：

```
http://mygardensite.org/products/4c4b1476238d3b4dd5003981
```

有意义的ID会更好一点：

```
http://mygardensite.org/products/wheel-barrow-9092
```

如果要为文档生成一个URL，我通常会建议增加一个短名称字段。这个字段应该有唯一性索引，这样就能把其中的值用作主键。假设将这个文档存储在products集合里，可以像下面这样创建唯一性索引：

```
db.products.ensureIndex({slug: 1}, {unique: true})
```

如果在slug上有唯一性索引，那么需要在插入产品文档时使用安全模式，这样就能得知插入成功与否。需要的话，可以换一个不同的短名称进行重试。举个例子，假设园艺商店里销售多种手推车，在开售新的手推车时，代码需要为新产品生成一个唯一的短名称。以下是在Ruby中执行插入的代码：

```
@products.insert({:name => "Extra Large Wheel Barrow",
          :sku  => "9092",
          :slug => "wheel-barrow-9092"},
          :safe => true)
```

这里需要重点说明的是指定了:safe => true。如果插入成功，没有抛出异常，表明选择了一个唯一的短名称。但如果抛出异常，代码就需要用一个新的短名称进行重试。

接下来，有一个名为details的键，指向包含不同产品详细信息的子文档，其中规定了重量、计重单位以及厂家的型号代码，你也可以存储其他特定属性。举例来说，如果在销售种子，可以在其中包含预期产量与收获时间；如果在销售割草机，可以包含马力、燃料类型和护根选项。details属性为这些动态属性提供了一个很好的容器。

请注意，还可以在同一个文档中存储产品的当前价格和历史价格。pricing键指向一个包含零售价和特价的对象。price_history则恰恰相反，指向一个价格数组。像这样存储文档副本是一种常见的版本化技术。

随后是一个产品标签名称的数组，在第1章里我们看到过类似的标签示例，这个技术值得反复演示。这是最简单、最佳的存储条目相关标签的途径，同时还能保证查询的高效性，因为可以索引数组键。

那么关系呢？我们可以使用富文档结构，比如子文档和数组，在单个文档中存储产品细节、价格和标签，但最终可能需要关联其他集合中的文档。开始时，我们会把产品关联到一个分类里，这种产品与分类之间的关系通常会表示为多对多关系，每个产品属于多个分类，而每个分类又能包含多个产品。在RDBMS中，我们会使用联结表表示这样的多对多关系。联结表在单个表中存储两个表间的所有关系引用。使用SQL的join可以发起一条查询，检索产品以及它的全部分类，反之亦然。

MongoDB不支持联结操作，因此需要一种不同的多对多策略。看看手推车的文档，你会发现一个名为category_ids的字段，其中包含一个对象ID的数组。每个对象ID都是一个指针，指向某个分类文档的_id字段。下面是一个演示用的分类文档。

代码清单4-2　分类文档

```
doc =
{  _id: new ObjectId("6a5b1476238d3b4dd5000048"),
   slug: "gardening-tools",
   ancestors: [{ name: "Home",
                 _id:  new ObjectId("8b87fb1476238d3b4dd500003"),
                 slug: "home"
                },

                { name: "Outdoors",
                  _id:  new ObjectId("9a9fb1476238d3b4dd5000001"),
                  slug: "outdoors"
                }
   ],

   parent_id: new ObjectId("9a9fb1476238d3b4dd5000001"),

   name: "Gardening Tools",
   description: "Gardening gadgets galore!",
}
```

如果回头看看产品文档，仔细观察category_ids字段里的对象ID，你会发现该产品关联了

刚才的Gardening Tools分类。在产品文档中放入category_ids数组键让那些多对多查询成为可能。举例来说，查询Gardening Tools分类里的所有产品，代码很简单：

```
db.products.find({category_ids => category['_id']})
```

要查询指定产品的所有分类，可以使用$in操作符，它类似于SQL的IN指令：

```
db.categories.find({_id: {$in: product['category_ids']}})
```

有了刚才描述的多对多关系，再来说说分类文档本身。你一定已经注意到了标准的_id、slug、name和description字段，它们都很直截了当，可是父文档数组的含义就不那么清楚了。为什么要用这么大的篇幅为每个文档冗余存储祖先分类？事实是分类总是被设想为有层级的，在数据库中表示这种层级的方式有很多种。[①]选择的策略总是依赖于应用程序的需求。本例中，由于MongoDB不支持关联查询，我们选择了去正规化，将上级分类的名称放入每个子分类的文档里。这样一来，查询Gardening Products分类时，就不需要执行额外的查询来获取上级分类（Outdoors和Home）的名称和URL了。

一些开发者可能会觉得这种级别的去正规化是不可接受的。还有其他方式可以用来表示树结构，附录B里就讨论了其中一种方式。但就目前而言，最佳的Schema是由应用程序需求决定的，无需受制于理论，试着接受各种可能性吧。在接下来的两章里你将看到更多对这种结构进行查询与更新的例子，其中的基本原理会变得越来越明朗。

4.2.2 用户与订单

看看如何对用户与订单建模，以此阐明另一种常见关系——一对多关系，就是说每个用户都有多张订单。在RDBMS中，会在订单表里使用外键；此处的惯例很相似。请看代码清单4-3。

代码清单4-3 电子商务订单，带有条目明细、价格和送货地址

```
doc =
{ _id: ObjectId("6a5b1476238d3b4dd5000048")
  user_id: ObjectId("4c4b1476238d3b4dd5000001")

  state: "CART",

  line_items:  [
    { _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku:  "9092",
      name:  "Extra Large Wheel Barrow",
      quantity:  1,
      pricing:  {
        retail:  5897,
        sale:  4897,
      }
    },

    { _id:  ObjectId("4c4b1476238d3b4dd5003981"),
      sku:  "10027",
```

① 在这篇MySQL开发者的文章里（http://mng.bz/83w4）介绍了两种方法——邻接列表和内嵌集合。

```
      name:  "Rubberized Work Glove, Black",
      quantity:  2,
      pricing: {
        retail:  1499,
        sale:  1299
      }
    }
  ],

  shipping_address:  {
     street: "588 5th Street",
     city: "Brooklyn",
     state: "NY",
     zip: 11215
   },

  sub_total:  6196
}
```

订单中的第二个属性`user_id`保存了一个用户的`_id`，它实际是一个指向示例用户（代码清单4-4中的用户，我们稍后会讨论这段代码）的指针。这一设计能方便地查询关系中的任意一方。要找到一个用户的所有订单非常简单：

```
db.orders.find({user_id: user['_id']})
```

要获取指定订单的用户同样很简单：

```
user_id = order['user_id']
db.users.find({_id: user_id})
```

像这样使用对象ID，能很方便地在订单与用户之间建立起一对多关系。

我们再来看看订单文档中的其他亮点。一般来说，我们会使用丰富的表示方式来承载文档数据模型，文档中既有订单条目明细又有送货地址。在正规化的关系型模型中，这些属性会被放在不同的数据表里。而这里，条目明细包含一个子文档数组，每个子文档都描述了购物车里的一个产品。送货地址属性指向一个对象，其中包含了地址信息。

让我们花点时间讨论一下这个表述的优点。首先，它易于人们理解，完整的订单概念都能被封装在一个实体里，包括条目明细、送货地址以及最终的支付信息。查询数据库时，可以通过一条简单的查询返回整个订单对象。其次，可以把产品在购买时的信息保存在订单文档里。最后，正如接下来的两章里会看到的，能轻而易举地查询并修改订单文档，这应该也是你能想到的。

用户文档也用了类似的模式，其中保存了一个地址文档的列表，还有一个支付方法文档的列表。此外，在文档的最上层还能找到任何用户模型里都有的基本常见属性。与产品的短名称字段一样，在用户名字段上添加了唯一索引。

代码清单4-4　用户文档，带有地址和支付方法

```
{ _id: new ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
```

```
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",

  addresses:  [
    {name: "home",
     street: "588 5th Street",
     city: "Brooklyn",
     state: "NY",
     zip: 11215},

    {name: "work",
     street: "1 E. 23rd Street",
     city: "New York",
     state: "NY",
     zip: 10010}
  ],

  payment_methods:  [
    {name: "VISA",
     last_four: 2127,
     crypted_number: "43f6ba1dfda6b8106dc7",
     expiration_date: new Date(2014, 4)
    }
  ]
}
```


4.2.3 评论

最后出场的示例数据模型是产品评论。一般而言，每个产品都会有多条评论，而该关系是用对象ID引用product_id来编码的，正如你在示例评论文档中看到的那样。

代码清单4-5 产品评论文档

```
{ _id: new ObjectId("4c4b1476238d3b4dd5000041"),
  product_id: new ObjectId("4c4b1476238d3b4dd5003981"),
  date: new Date(2010, 5, 7),
  title: "Amazing",
  text: "Has a squeaky wheel, but still a darn good wheel barrow.",
  rating: 4,

  user_id: new ObjectId("4c4b1476238d3b4dd5000041"),
  username: "dgreenthumb",


  helpful_votes: 3,
  voter_ids: [ new ObjectId("4c4b1476238d3b4dd5000041"),
                new ObjectId("7a4f0376238d3b4dd5000003"),
                new ObjectId("92c21476238d3b4dd5000032")
  ]
}
```

大多数剩余属性的含义都不言而喻。我们存储了评论的日期、标题和内容、用户的评分，以及用户的ID。有些意外的是还存储了用户名。毕竟，如果是RDBMS，可以通过关联用户表来获

取用户名。但因为在MongoDB中没有关联查询，所以有两个可选方案：针对每条评论再去查询一次用户集合，或者是接受去正规化。当所查询的属性（用户名）极有可能不会改变时，针对每条评论发起一次查询会很浪费。诚然，我们可以选择正规化的做法，通过两次MongoDB查询来显示所有的评论，但这里正在为常见情况设计Schema。因为修改用户名时需要在每个出现用户名的地方都做修改，这意味着修改用户名的代价更高了。但它的发生频率非常低，这足以让这种做法成为一个合理的设计选择。

另一点值得注意的地方是在评论文档里保存了投票信息。用户通常能对评论进行投票，这里在投票者ID数组中保存了每个投票用户的对象ID，这能避免用户对同一评论多次投票，同时也让我们有能力查询某个用户投过票的所有评论。注意，这里还缓存了有用投票的总数，以便能基于有用程度对评论进行排序。

目前，我们已经覆盖基本的电子商务数据模型了。如果这是你第一次接触MongoDB数据模型，那么要对其实用程度有所期待还是需要一定信心的。接下来的两章里会详细探讨该模型中剩下的东西，包括不重复地添加投票、修改订单、智能地查询产品，借此分别阐述查询与更新。

4.3　具体细节：数据库、集合与文档

我们暂时将电子商务示例放在一边，来看看数据库、集合与文档的核心细节。其中很多内容涉及了定义、特殊特性和极端情况。如果想知道MongoDB是如何分配数据文件的、文档中严格限制了哪些数据类型、使用固定集合有什么好处，请继续读下去。

4.3.1　数据库

数据库是集合的逻辑与物理分组。本节里，我们会讨论创建与删除数据库的细节。还会深入探讨MongoDB是如何在文件系统上为每个数据库分配空间的。

1. 管理数据库

MongoDB里没有显式创建数据库的方法，在向数据库中的集合写入数据时会自动创建该数据库。看看下面这段Ruby代码：

```
@connection = Mongo::Connection.new
@db = @connection['garden']
```

假定之前数据库并不存在，在执行这段代码之后仍然不会在磁盘上创建数据库。此处只是实例化了一个`Mongo::DB`类的实例。只有在向某个集合写入数据时才会创建数据文件。接下来：

```
@products = @db['products']
@products.save({:name => "Extra Large Wheel Barrow"})
```

调用`products`集合的`save`方法时，驱动会告诉MongoDB将产品文档插入到`garden.products`命名空间里。如果该命名空间并不存在，则会进行创建；其中还涉及在磁盘上分配`garden`数据库。

要删除数据库，意味着删除其中所有的集合，我们要发出一条特殊的命令。在Ruby里可以这

样删除garden数据库：

```
@connection.drop_database('garden')
```

在MongoDB Shell里，可以运行dropDatabase()方法：

```
use garden
db.dropDatabase();
```

在删除数据库时要格外小心，因为这个操作是无法撤销的。

2. 数据文件与空间分配

在创建数据库时，MongoDB会在磁盘上分配一组数据文件，所有集合、索引和数据库的其他元数据都保存在这些文件里。数据文件都被放置在启动mongod时指定的dbpath里。在未指定dbpath时，mongod会把文件全保存在/data/db里。让我们看看在创建了garden数据库后/data/db目录里的情况：

```
$ cd /data/db
$ ls -al
drwxr-xr-x  6 kyle  admin        204 Jul 31 15:48 .
drwxrwxrwx  7 root  admin        238 Jul 31 15:46 ..
-rwxr-xr-x  1 kyle  admin   67108864 Jul 31 15:47 garden.0
-rwxr-xr-x  1 kyle  admin  134217728 Jul 31 15:46 garden.1
-rwxr-xr-x  1 kyle  admin   16777216 Jul 31 15:47 garden.ns
-rwxr-xr-x  1 kyle  admin          6 Jul 31 15:48 mongod.lock
```

先来看mongod.lock文件，其中存储了服务器的进程ID。[①]数据库文件本身是依据所属的数据库命名的。garden.ns是第一个生成的文件。文件扩展名*ns*表示*namespaces*，意即命名空间。数据库中的每个集合和索引都有自己的命名空间，每个命名空间的元数据都存放在这个文件里。默认情况下，.ns文件大小固定在16 MB，大约可以存储24 000个命名空间。也就是说数据库中的索引和集合总数不能超过24 000。我们几乎不可能使用这么多集合与索引，但如果真有需要，可以使用--nssize服务器选项让该文件变得更大一点。

除了创建命名空间文件，MongoDB还为集合与索引分配空间，就在以从0开始的整数结尾的文件里。查看目录的文件列表，会看到两个核心数据文件，64 MB的garden.0和128 MB的garden.1。这些文件的初始大小经常会让新用户大吃一惊，但MongoDB倾向于这种预分配的做法，这能让数据尽可能连续存储。如此一来，在查询和更新数据时，这些操作能更靠近一点，而不是分散在磁盘各处。

在向数据库添加数据时，MongoDB会继续分配更多的数据文件。每个新数据文件的大小都是上一个已分配文件的两倍，直到达到预分配文件大小的上限——2 GB，即garden.2会是256 MB，garden.3是512 MB，以此类推。此处基于这样一个假设，如果总数据大小呈恒定速率增长，应该逐渐增加数据文件分配的空间，这是一种相当标准的分配策略。当然，这么做的后果之一就是分

① 永远不要删除或修改锁定文件，除非是在对非正常关闭的数据库进行恢复。如果在启动mongod时弹出一个与锁定文件有关的错误消息，很有可能是之前没有正常关闭，可能需要初始化一个恢复进程。我们会在第10章里进一步讨论该话题。

配的空间与实际使用的空间之间会存在很大的差距[①]。

可以使用stats命令检查已使用空间和已分配空间：

```
> db.stats()
{
  "collections" : 3,
  "objects" : 10004,
  "avgObjSize" : 36.005,
  "dataSize" : 360192,
  "storageSize" : 791296,
  "numExtents" : 7,
  "indexes" : 1,
  "indexSize" : 425984,
  "fileSize" : 201326592,
  "ok" : 1
}
```

在这个例子里，fileSize字段标明了为该数据库分配的文件空间的总和，就是简单地把garden数据库的两个数据文件（garden.0和garden.1）的大小加起来。比较有意思的是dataSize和storageSize两者的差值，前者是数据库中BSON对象的实际大小，后者包含了为集合增长预留的额外空间和未分配的已删除空间。[②]最后，indexSize的值是数据库索引大小的总合。关注总计索引大小是很重要的，当所有用到的索引都能放入内存时，数据库的性能是最好的。我将在第7章和第10章里介绍排查性能问题的技术时详细讨论这个话题。

4.3.2　集合

集合是结构上或概念上相似的文档的容器。本节会更详细地描述集合的创建与删除。随后，我会介绍MongoDB特有的固定集合，并给出一些例子，演示核心服务器内部是如何使用集合的。

1. 管理集合

正如在上一节里看到的，在向一个特定命名空间中插入文档时还隐式地创建了集合。但由于存在多种集合类型，MongoDB还提供了创建集合的命令。在Shell中可以执行：

```
db.createCollection("users")
```

在创建标准集合时，有选项能指定预先分配多少字节的存储空间。方法如下（但通常没必要这么做）：

```
db.createCollection("users", {size: 20000})
```

集合名里可以包含数字、字母或.符号，但必须以字母或数字开头。在MongoDB内部，集合

① 这在空间很宝贵的部署环境下会带来一些问题，针对此类情况，可以组合使用--noprealloc和--smallfiles这两个服务器选项。

② 严格说来，集合就是每个数据文件里按块分配的空间，这些块称为区段（extent）。storageSize就是为集合区段所分配空间的总额。

名是用它的命名空间名称来标识的，其中包含了它所属的数据库的名称。因此，严格说起来，在往来于核心服务器的消息里引用产品集合时应该用garden.products。这个完全限定集合名不能超过128个字符。

有时在集合名里包含.符号很有用，它能提供某种虚拟命名空间。举例来说，可以想象有一系列集合使用了下列名称：

```
products.categories
products.images
products.reviews
```

请牢记这只是一种组织上的原则，数据库对名字里带有.的集合和其他集合是一视同仁的。

我之前已经提到过从集合中删除文档和彻底删除集合了，现在你还需要知道集合是可以重命名的。比如，可以用Shell里的renameCollection方法重命名产品集合：

```
db.products.renameCollection("store_products")
```


2. 固定集合

除了目前为止创建的标准集合，我们还可以创建*固定集合*（capped collection）。固定集合原本是针对高性能日志场景设计的。它们与标准集合的区别在于其大小是固定的，也就是说，一旦固定集合到达容量上限，后续的插入会覆盖集合中最先插入的文档。在只有最近的数据才有价值的情况下，这种设计免除了用户手工清理集合的烦恼。

要理解如何使用固定集合，可以假设想要追踪访问我们站点的用户的行为。此类行为会包含查看产品、添加到购物车、结账与购买。可以写个脚本来模拟向固定集合记录这些用户行为的日志记录功能。在这个过程里，我们会看到这些集合的一些有趣属性。下面是一个示例。

代码清单4-6　模拟向固定集合中记录用户行为日志

```
require 'rubygems'
require 'mongo'

VIEW_PRODUCT = 0
ADD_TO_CART  = 1
CHECKOUT     = 2
PURCHASE     = 3

@con = Mongo::Connection.new
@db  = @con['garden']

@db.drop_collection("user.actions")

@db.create_collection("user.actions", :capped => true, :size => 1024)
@actions = @db['user.actions']

20.times do |n|
  doc = {
    :username => "kbanker",
    :action_code => rand(4),
    :time => Time.now.utc,
```

```
    :n => n
  }

  @actions.insert(doc)
end
```

首先，使用DB#create_collection方法[1]创建一个名为users.actions、大小为1 KB的固定集合。接下来，插入20个示例日志文档。每个文档都包含用户名、动作代码（存储内容为0~3的整数）和时间戳，还要加入一个不断增加的整数*n*，这样就能标识出哪个文档过期了。现在从Shell里查询集合：

```
> use garden
> db.user.actions.count();
10
```

尽管插入了20个文档，但集合里却只有10个文档，查询一下集合内容，你就能知道为什么了：

```
db.user.actions.find();
{ "_id" : ObjectId("4c55f6e0238d3b201000000b"), "username" : "kbanker",
  "action_code" : 0, "n" : 10, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000c"), "username" : "kbanker",
  "action_code" : 4, "n" : 11, "time" : "Sun Aug 01 2010 18:36:16" }
{ "_id" : ObjectId("4c55f6e0238d3b201000000d"), "username" : "kbanker",
  "action_code" : 2, "n" : 12, "time" : "Sun Aug 01 2010 18:36:16" }
...
```

返回的文档是按照插入顺序排列的。仔细观察n的值，很明显，集合中最老的文档是第十个插入的文档，也就是说文档0~9都已经过期了。既然该固定集合最大是1024字节，仅包含10个文档，也就是说每个文档大致是100字节。后面你将看到如何验证这个假设。

在此之前，我要再指出固定集合与标准集合之间的几个不同点。固定集合默认不为_id创建索引，这是为了优化性能，没有索引，插入会更快。如果实在需要_id索引，可以手动构建索引。在不定义索引的情况下，最好把固定集合当做用于顺序处理的数据结构，而非用于随机查询的数据结构。为此，MongoDB提供了一个特殊的排序操作符，按自然插入顺序[2]返回集合的文档。之前的查询是按自然顺序正向输出结果的，如果要逆序输出，必须使用$natural排序操作符：

```
> db.user.actions.find().sort({"$natural": -1});
```

除了按自然顺序排列文档，并放弃索引，固定集合还限制了CRUD操作。比如，不能从固定集合中删除文档，也不能执行任何会增加文档大小的更新操作。[3]

3. 系统集合

MongoDB内部对集合的使用方式可以体现它的部分设计思想，system.namespaces与system. indexes就属于这些特殊系统集合。前者可以查询到当前数据库中定义的所有命名空间：

① Shell里的等效创建命令是db.createCollection("users.actions", {capped: true, size: 1024})。
② 自然顺序是文档保存在磁盘上的顺序。
③ 因为固定集合最早是为日志记录功能而设计的，不需要实现删除或更新文档功能，这些功能会让负责旧文档过期的代码复杂化。去掉这些功能，固定集合获得了设计的简单性和高效性。

```
> db.system.namespaces.find();
{ "name" : "garden.products" }
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.user.actions", "options" :
    { "create": "user.actions", "capped": true, "size": 1024 } }
```

后者存储了当前数据库的所有索引定义。要获取garden数据库的索引，查询该集合即可：

```
> db.system.indexes.find();
{ "name" : "_id_", "ns" : "garden.products", "key" : { "_id" : 1 } }
```

system.namespaces与system.indexes都是标准的集合，但MongoDB使用固定集合来做复制。每个副本集的成员都会把所有的写操作记录到一个特殊的oplog.rs固定集合里。从节点顺序读取这个集合的内容，再把这些新操作应用到自己的数据库里。第9章将更详细地讨论这个系统集合。

4.3.3 文档与插入

我们将通过讨论文档及其插入的细节来结束这章。

1. 文档序列化、类型和限制

正如上一章中说的那样，所有文档在发送到MongoDB之前都必须序列化成BSON；随后再由驱动将文档从BSON反序列化到语言自己的文档表述。大多数驱动都提供了一个简单的接口，可以进行BSON的序列化和反序列化。我们可能会需要查看发送给服务器的内容，因此了解这部分功能在驱动中是如何实现的会非常有用。举例来说，前文在演示固定集合，我们有理由假设示例文档的大小大约是100字节。可以通过Ruby驱动的BSON序列化器来验证这一假设：

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}

bson = BSON::BSON_CODER.serialize(doc)

puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

serialize方法会返回一个字节数组。如果运行上述代码，会得到一个82字节的BSON对象，和我们估计的差不多。如果想要在Shell里检查BSON对象的大小，可以这样做：

```
> doc = {
  _id: new ObjectId(),
  username: "kbanker",
  action_code: Math.ceil(Math.random() * 5),
  time: new Date(),
  n: 1
}

> Object.bsonsize(doc);
82
```

同样也是82字节。82字节的文档大小和100字节的估计值的差别在于普通集合和文档的开销。

反序列化BSON也很简单，可以尝试运行以下代码：

```
deserialized_doc = BSON::BSON_CODER.deserialize(bson)

puts "Here's our document deserialized from BSON:"
puts deserialized_doc.inspect
```

请注意，不是所有Ruby散列都能被序列化。要正确序列化，键名必须是合法的，每个值都必须能转换为BSON类型。合法的键名由`null`结尾的字符串组成，最大长度为255字节。字符串可以包含任意ASCII字符的组合，但有三种情况例外：不能以`$`开头，不能包含`.`字符，除了结尾处外不能包含`null`字节。在Ruby里，可以用符号充当散列的键，在序列化时它们会被转换为等效的字符串。

应该慎重选择键名的长度，因为这是存储在文档里面的。这种做法与RDBMS截然不同，RDBMS里列名总是与数据行分开保存的。因此，在使用BSON时，可以用`dob`代替`date_of_birth`作为键名，这样一来每个文档都能省下10字节。这个数字听起来并不大，但一旦有了10亿个文档，这个更短的键名能帮我们省下将近10 GB的存储空间。但这也不是让你肆意缩短键名长度，请选择一个合适的键名。如果有大量的数据，更“经济”的键名能帮助省下不少空间。

除了合法的键名，文档还必须包含可以序列化为BSON的值。在http://bsonspec.org可以找到一张BSON类型的表格，其中有示例和注解。此处我只会指出一些重点和容易碰到的陷阱。

- 字符串

所有字符串都必须编码为UTF-8，虽然UTF-8就快成为字符编码的行业标准了，但还是有很多地方仍在使用旧的编码。在将数据从遗留系统导入到MongoDB时用户通常会遇到一些问题。解决方案一般是在插入前将内容转换为UTF-8，或者将文本保存为BSON二进制类型。[①]

- 数字

BSON规定了三种数字类型：`double`、`int`和`long`。也就是说BSON可以编码各种IEEE浮点数值，以及各种8字节以内的带符号整数。在动态语言里序列化整数时，驱动会自己决定是将其序列化为`int`还是`long`。实际上，只有一种常见情况需要显式地决定数字类型，那就是通过JavaScript Shell插入数字数据时。很遗憾，JavaScript天生就支持一种数字类型，即`Number`，它等价于IEEE的双精度浮点数。因此，如果希望在Shell里将一个数字保存为整数，需要使用`NumberLong()`或`NumberInt()`显式指定。试试下面这段代码：

```
db.numbers.save({n: 5});
db.numbers.save({ n: NumberLong(5) });
```

这里向`numbers`集合添加了两个文档，虽然两个值是一样的，但第一个被保存成了双精度浮

---

① 顺便说一下，如果你还不太了解字符编码，推荐你读一下Joel Spolsky那篇著名的介绍字符编码的文章，参见http://mng.bz/LVO6。如果你是一名Ruby爱好者，也许还会想读一读James Edward Gray关于Ruby 1.8和1.9字符编码的一系列文章，参见http://mng.bz/wc4J。

点数，第二个则被保存成了长整数。查询所有n是5的文档会将这两个文档一并返回：

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

但是可以看到第二个值被标记为长整数。另一种做法是使用特殊的$type操作符来查询BSON类型。每种BSON类型都由一个从1开始的整数来标识。如果查看http://bsonspec.org上的BSON规范，会看到双精度浮点数是类型1，而64位整数是类型18。所以，可以根据类型来查询集合的值：

```
> db.numbers.find({n: {$type: 1}});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }

> db.numbers.find({n: {$type: 18}});
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

这也证实了两者在存储上的不同。在生产环境里我们几乎用不上$type操作符，但在调试时，这是个很棒的工具。

4

另一个和BSON数字类型有关的问题是其中缺乏对小数的支持。这意味着在MongoDB中保存货币值时需要使用整数类型，并且以美分为单位来保存货币值。

● 日期时间

BSON的日期时间类型是用来存储时间的，用带符号的64位整数来标识Unix epoch[①]毫秒数，采用的时间格式是UTC（Coordinated Universal Time，协调世界时）。负值代表时间起点之前的毫秒数。

以下是一些使用时的注意事项。首先，如果在JavaScript里创建日期，请牢记JavaScript日期里的月份是从0开始的。也就是说new Date(2011, 5, 11)创建出的日期对象表示2011年6月11日。其次，如果使用Ruby驱动存储时间数据，BSON序列化器会期待传入一个UTC格式的Ruby Time对象。其结果就是不能使用包含时区信息的日期类，因为BSON 日期时间无法对它进行编码。

● 自定义类型

如果希望连同时区一起保存时间该怎么办呢？有时候光有基本的BSON类型是不够的。虽然无法创建自定义BSON类型，但可以结合几个不同的原生BSON值，以此创建自己的虚拟类型。举例来说，想要保存时区和时间，可以使用这样一种文档结构，Ruby代码如下：

```
{:time_with_zone =>
  {:time => Time.utc.now,
   :zone => "EST"
  }
}
```

要编写一个能透明处理此类组合表述的应用程序并不复杂。真实情况往往就是这样的。例如，MongoMapper（用Ruby编写的MongoDB对象映射器）允许为任意对象定义to_mongo和

① Unix epoch是从1970年1月1日午夜开始的协调世界时。

from_mongo方法，方便此类自定义组合类型的使用。

● 文档大小的限制

MongoDB v2.0中BSON文档的大小被限制在16 MB①。出于两个原因需要增加这个限制，首先是为了防止开发者创建难看的数据模型。虽然在这个限制下仍然会有差劲的数据模型，但16 MB的限制还是有帮助的，尤其是能避免深层次的嵌套，这种嵌套对于MongoDB的新手是个常见的数据建模问题。深层嵌套的文档很难使用，最好能将它们展开到各自不同的集合里。

第二个原因与性能有关，在服务器端查询大文档，在将结果发送个客户端之前需要将文档复制到缓冲区里。这个复制动作的代价可能很大，尤其在客户端并不需要整个文档时（这种情况很常见）。②此外，一旦发送之后，就会在网络中传输这些文档，驱动还要对其进行反序列化。如果一次请求大量MB数量级的文档，这笔开销会极大。

结论就是，如果有很大的对象，也许可以将它们拆开，修改其数据模型，使用一到两个额外的集合。如果仅仅存储大的二进制对象，比如图片或视频，这又是另一种情况，附录C里有与处理大型二进制对象相关的内容。

2. 批量插入

在有了正确的文档之后，就该执行插入操作了。第3章里已经讨论了很多与插入相关的细节，包括生成对象ID、网络层上插入是如何实现的，还有安全模式。但还有一个特性值得探讨，那就是批量插入。

所有的驱动都可以一次插入多个文档，这在有很多数据需要插入时非常有用，比如初始化批量导入或者从另一个数据库系统迁移数据时。回想之前向user.actions集合插入20个文档的例子，如果再去读下代码，会发现每次只插入一个文档。使用下面的代码，事先构造一个40个文档的数组，随后将整个文档数组传递给insert方法：

```
docs = (0..40).map do |n|
  { :username => "kbanker",
    :action_code => rand(5),
    :time => Time.now.utc,
    :n => n
  }
end
@col = @db['test.bulk.insert']
@ids = @col.insert(docs)

puts "Here are the ids from the bulk insert: #{@ids.inspect}"
```

与单独返回一个对象ID有所不同，批量插入会返回所有插入文档的对象ID数组。用户经常会问，理想的批量插入数量是多少？答案受到太多具体因素的影响，理想的数字范围为10~200。在具体情况中，基准测试的结果是最有价值的。数据库方面唯一的限制是单次插入操作不能超过16 MB

① 这个数字在各个服务器版本之间有所不同，而且还在继续增加。要了解正在使用的服务器版本对应的限制值，可以在Shell里运行db.ismaster，查看maxBsonObjectSize字段。如果没有这个字段，那么该限制是4 MB（正在使用一个非常古老的MongoDB版本）。

② 在下一章里会看到，可以指定查询返回文档的哪些字段，以此控制响应的大小。如果经常这么做，就可以重新考虑一下你的数据模型了。

上限。经验表明大多数高效的批量插入都远低于该限制。

4.4　小结

这一章涉及了很多基础的内容；为自己的所得欢呼吧！

开始时，我们探讨了理论上的Schema设计，随后为一个电子商务应用程序设计了大致的数据模型，让你有机会了解生产系统中的文档是什么样的，而且以更具体的方式来思考RDBMS与MongoDB之间Schema的区别。

本章结尾处我们详细了解了与数据库、文档和集合相关的内容；你以后也会参考本章的内容。我已经阐述了MongoDB的入门知识，但还没真正开始接触数据。下一章中一切都会有所不同，我们将会看到即时查询的威力。

第 5 章

查询与聚合

5

本章内容

- ❑ 查询电子商务数据模型
- ❑ 详细解说 MongoDB 查询语言
- ❑ 使用 MapReduce 和分组进行聚合

MongoDB中使用的不是SQL，而是它自己的查询语言，与JSON很相似。贯穿全书，我们都在探讨这门查询语言，但本章我们要接触一些真实示例。注意，我们将回顾上一章里介绍的电子商务数据模型，基于它进行很多不同的查询，包括_id查找、范围查询、排序和投影（projection）。我们还将纵览MongoDB查询语言，详细介绍每个查询操作符。

除了查询，本章还会涉及聚合（aggregation）这个主题。查询允许你获得存储的数据，聚合函数则能汇总并重新组织那些数据。首先，我们通过本书的电子商务示例数据集了解如何进行聚合，此处会关注MongoDB的分组和MapReduce函数。随后，我会给出这些函数的完整说明。

请牢记，本章中看到的MongoDB查询语言和聚合函数仍在不断完善之中，每个版本都会有所改进。照目前的情况来看，掌握MongoDB中的查询与聚合并不是了解其中的具体细节，而是找到完成日常任务的最佳途径。通过本章的示例，我会为你指出一条“明路”。到本章结束时，你应该已经能很好地理解MongoDB中的查询与聚合了，而且能将它们运用到应用程序Schema的设计之中。

5.1 电子商务查询

本节继续探讨上一章中给出的电子商务数据模型。我们已经为产品、分类、用户、订单和产品评论定义了文档结构，有了这一结构，让我们来看看如何在一个典型的电子商务应用程序里查询这些实体。其中的一些查询非常简单，举例来说，_id查找应该毫无秘密可言。但我们还会看到一些较复杂的模式，包括查询并显示分类层级，以及为产品列表提供过滤视图。除此之外，要将效率问题牢记于心，针对这些查询还要寻找可能的索引。

5.1.1 产品、分类与评论

大多数电子商务应用程序都提供至少两种基本的产品和分类视图。第一种是产品主页，突出

某个指定的产品，显示其评论，给出一些与产品分类相关的信息。第二种是产品列表页面，允许用户浏览分类层级，查看所选分类中所有产品的缩略图。让我们先从产品主页入手，多数情况下这是两者中比较容易的一个。

假设产品页面URL是以产品的短名称作为键的，这样就能通过以下三个查询获得产品页面中所需的所有信息：

```
db.products.findOne({'slug': 'wheel-barrow-9092'})
db.categories.findOne({'_id': product['main_cat_id']})
db.reviews.find({'product_id': product['_id']})
```

第一个查询通过短名称wheel-barrow-9092找到了产品。一旦有了产品，就能从categories集合里用简单的_id查询找到其分类信息。最后，再发起一次简单查询，获得与该产品相关的所有评论。

相信你已经注意到了，头两个查询用的是findOne方法，但最后一个查询却用了find方法。所有的MongoDB驱动都提供了这两个方法，很有必要温习一下两者的区别。正如第3章中所说的那样，find返回的是游标对象，而findOne返回的是一个文档。上面用到的findOne和下面这条语句是等价的：

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

如果仅仅想要一个文档，只要它存在，findOne就能返回它。如果需要返回多个文档，就需要使用find了，该方法会返回一个游标，你需要在应用程序里对它进行迭代。

5

现在再来看看产品页面的查询，还有什么问题吗？如果觉得评论的查询有点粗放，那就对了。该查询会返回指定产品的所有评论，但这种做法在产品拥有成百上千条评论时显然不够严谨。大多数应用程序都会对评论进行分页，为此，MongoDB提供了skip和limit选项。可以像下面这样用它们对评论文档进行分页：

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)
```

如果还希望以一致的顺序显示评论，就需要对查询结果进行排序。如果想要按照每条评论收到的投票数排序，方法很简单：

```
db.reviews.find({'product_id': product['id']}).sort(
                {helpful_votes: -1}).limit(12)
```

简而言之，这条查询告诉MongoDB按照投票总数降序排列，返回前12条评论。有了skip、limit和sort，只需在开始时决定是否需要分页。为此，可以发起一次count查询。随后结合count的结果和想要的评论页码再进行查询。完整的产品页面查询是这样的：

```
product  = db.products.findOne({'slug': 'wheel-barrow-9092'})
category = db.categories.findOne({'_id': product['main_cat_id']})
reviews_count = db.reviews.count({'product_id': product['_id']})
reviews = db.reviews.find({'product_id': product['_id']}).
                           skip((page_number - 1) * 12).
                           limit(12).
                           sort({'helpful_votes': -1})
```

这些查询语句都应该使用索引。因为短名称也可以当做主键来用，所以应该为它们加上唯一

性索引。而且你应该也知道所有标准集合的_id字段都会自动加上唯一性索引，对于任何充当引用的字段也都应该为它们加上索引。在本例中，这些字段还包括评论集合中的user_id和product_id字段。

完成了产品主页的查询，现在可以将视线转向产品列表页面了。此类页面会展现一个指定的分类，页面中带有可浏览的产品列表，还有指向上级分类和同级分类的链接。

产品列表页面是根据产品分类来定义的，因此针对该页面的请求将使用分类的短名称：

```
category = db.categories.findOne({'slug': 'outdoors'})
siblings = db.categories.find({'parent_id': category['parent_id']})
products = db.products.find({'category_id': category['_id']}).
                            skip((page_number - 1) * 12).
                            limit(12).
                            sort({average_review: -1})
```

同级分类是指拥有相同parent_id的其他分类，因此对它的查询非常简单。既然产品都包含一个分类ID的数组，那么查询指定分类里的所有产品也同样很简单。还是需要使用与之前评论相同的分页模式，不同的只是按照平均产品评分进行排序，我们还可以提供其他排序方法（根据名称、价格等），改变排序字段即可。[①]

产品列表页面还有一种基本情况，就是查询顶级分类，没有产品。只需在分类集合中查找parent_id是nil的分类就可以了：

```
categories = db.categories.find({'parent_id': nil})
```

5.1.2 用户与订单

上一节里的查询仅限于_id查找和排序，对于用户与订单，由于希望为订单生成基本的报表，我们的查询会更进一步。

先从稍微简单一些的查询入手：用户身份验证。用户提供用户名和密码登录到应用程序中，因此经常会使用以下查询：

```
db.users.findOne({username: 'kbanker',
  hashed_password: 'bd1cfa194c3a603e7186780824b04419'})
```

如果用户存在且密码正确，会返回完整的用户文档；否则就没有返回结果。这条查询是可接受的。但如果要考虑性能，可以只返回_id字段，用它就能发起会话了。毕竟在用户文档里保存了地址、支付方法和其他诸多个人信息。如果需要的只是一个字段，又何必在网络上传输那些数据，并在驱动端反序列化它们呢？可以通过投影来限制返回的字段：

```
db.users.findOne({username: 'kbanker',
  hashed_password: 'bd1cfa194c3a603e7186780824b04419'},
  {_id: 1})
```

① 考虑这些排序是否高效是很重要的。可以依靠索引来处理排序，但随着排序选项的增加，索引数量也会相应增加，维护这些索引的成本就可能超出可接受的范围。如果每个分类的产品数量很少，这种情况尤为突出。我们将在第8章中深入讨论这一话题，但你可以先考虑起来了。

现在的响应里只有文档的_id字段了：

```
{ _id: ObjectId("4c4b1476238d3b4dd5000001") }
```

还有很多其他对用户集合users的查询。举例来说，你有一个管理后台，允许根据不同条件查询用户。通常会查询某个字段，比如last_name：

```
db.users.find({last_name: 'Banker'})
```

这条查询可以执行，但仅限于精确匹配的场景。也许你并不知道如何拼写某个用户的名字，这时就需要部分匹配的查询。假设知道用户的姓氏是以*Ba*开头的，在SQL里可以使用LIKE条件来进行查询：

```
SELECT * from users WHERE last_name LIKE 'Ba%'
```

MongoDB中语义上与其等价的是一个正则表达式：

```
db.users.find({last_name: /^Ba/})
```

和RDBMS一样，像这样的前缀搜索可以用上索引。[①]

在面向用户进行市场营销之前，可能希望明确用户范围，举例来说，想要获得所有居住在Upper Manhattan[②]的用户，可以针对用户的邮政编码发起范围查询：

```
db.users.find({'addresses.zip': {$gte: 10019, $lt: 10040}})
```

每个用户文档都包含一个地址数组，其中有一到多个地址。如果这些地址中有哪个邮政编码落在指定的范围里，那么这个用户文档就会被匹配到。要让该查询更高效，可以在address.zip上定义一个索引。

根据地域来寻找目标用户未必是提升转化率的最好途径，根据用户买过的东西来进行分组会更有意义。这会要求执行两步查询：首先，基于特定产品获得一个订单集合，一旦有了订单，就能查询关联的用户了。[③]假设想找到所有购买过大型手推车的用户，可以使用MongoDB的点符号深入line_items数组，查询指定SKU：

```
db.orders.find({'line_items.sku': "9092")
```

还可以针对结果集做限制，将订单限定在某个时间段里。只需简单地添加一个查询条件，指定最小的订单日期：

```
db.orders.find({'line_items.sku': "9092",
    'purchase_date': {$gte: new Date(2009, 0, 1)}})
```

如果这些查询很频繁，需要一个复合索引，先按照SKU排序，然后再按照购买日期排序。可以像下面这样创建索引：

```
db.orders.ensureIndex({'line_items.sku': 1, 'purchase_date': 1})
```

① 如果不熟悉正则表达式，请注意：正则表达式/^Ba/可以解读为“行首以*B*打头随后是*a*”。

② 曼哈顿上城，指纽约市曼哈顿的北部区域。——译者注

③ 如果之前用过关系型数据库，此处无法对订单和用户表进行关连查询可能会让你觉得不便，但大可不必如此，在MongoDB里执行这样的客户端关联是很常见的。

在查询orders集合时，所寻找的就是用户ID的列表。因此，使用投影会更高效一些。下面这段代码中，先规定只要user_id字段，然后将查询结果转换为一个简单的ID数组，随后再用$in操作符查询users集合：

```
user_ids = db.orders.find({'line_items.sku': "9092",
  purchase_date: {'$gt': new Date(2009, 0, 1)}},
  {user_id: 1, _id: 0}).toArray().map(function(doc) { return doc['user_id'] })
users = db.users.find({_id: {$in: user_ids}})
```

在ID数组有上千个元素时，这种使用ID数组和$in来查询集合的做法会更高效。对于更大的数据集，比如有100万用户购买了手推车，最好是将那些用户ID写到临时集合中，然后再顺序查询。

在下一章里你会看到更多针对该数据的查询，还会了解到如何用MongoDB的聚合函数分析数据。但为了充实你的知识，接下来要深入介绍MongoDB的查询语言，特别是说明其中每个操作符的语法。

5.2　MongoDB 查询语言

是时候了解MongoDB那无与伦比的查询语言了，我会先从查询的描述、语义和类型开始讲述，然后讨论游标，因为每条MongoDB查询本质上来看都是实例化了一个游标并获取它的结果集。掌握了这些基础知识之后，我再分类介绍MongoDB查询操作符。[①]

5.2.1　查询选择器

我们先大致了解一下查询选择器，尤其要关注所有能用它们表示的查询类型。

1. 选择器匹配

要指定一条查询，最简单的方法就是使用选择器，其中的键值对直接匹配要找的文档。下面是两个例子：

```
db.users.find({last_name: "Banker"})
db.users.find({first_name: "Smith", age: 40})
```

第二条查询的意思是“查找所有first_name是Smith，并且age是40的用户”。请注意，无论传入多少个键值对，它们必须全部匹配；查询条件之间相当于用了布尔运算符AND。如果想要表示布尔运算符OR，可以阅读后面关于布尔操作符的部分。

2. 范围查询

我们经常需要查询某些值在一个特定范围内的文档。在SQL中，可以使用<、<=、>和>=；在MongoDB中有类似的一组操作符$lt、$lte、$gt和$gte。贯穿全书，我们都在使用这些操作符，它们的行为与预期的一样。但初学者在组合使用这些操作符时偶尔会很费力，常见的错误是重复搜索键：

```
db.users.find({age: {$gte: 0}, age: {$lte: 30}})
```

① 除非你十分关心细节，否则在初次阅读时可以跳过这部分内容。

因为同一文档中同一级不能有两个相同的键，所以这个查询选择器是无效的，两个范围操作符只会应用其中之一。可以用下面的方式来表示该查询：

```
db.users.find({age: {$gte: 0, $lte: 30}})
```

还有一个值得注意的地方：范围操作符涉及了类型。仅当文档中的值与要比较的值类型相同时[①]，范围查询才会匹配该值。例如，假设有一个集合，其中包含以下文档：

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

然后执行如下查询：

```
db.items.find({value: {$gte: 97}})
```

你可能觉得这条查询应该把六个文档全部返回，因为那几个字符串在数值上跟整数97、98和99是等价的。但事实并非如此，该查询只会返回整数结果。如果想让结果是字符串，就应该改用字符串来进行查询：

```
db.items.find({value: {$gte: "a"}})
```

只要同一集合中永远不会为同一个键保存多种类型，就可以不用担心这条类型限制。这是一个很好的实践，你应该遵守它。

5

3. 集合操作符

`$in`、`$all`和`$nin`这三个查询操作符接受一到多个值的列表，将其作为谓词。如果任意给定值匹配搜索键，`$in`就返回该文档。我们可以使用该操作符返回所有属于某些离散分类集的产品。请看以下分类ID列表：

```
[ObjectId("6a5b1476238d3b4dd5000048"),
 ObjectId("6a5b1476238d3b4dd5000051"),
 ObjectId("6a5b1476238d3b4dd5000057")
]
```

如果它们分别对应割草机、手持工具和工作服分类，可以像下面这样查询所有属于这些分类的产品：

```
db.products.find({main_cat_id: { $in:
                 [ObjectId("6a5b1476238d3b4dd5000048"),
                  ObjectId("6a5b1476238d3b4dd5000051"),
                  ObjectId("6a5b1476238d3b4dd5000057") ] } } )
```

也可以把`$in`操作符想象成对单个属性的布尔运算符OR，之前的查询可以解释为“查找所有分类是割草机或手持工具或工作服的产品”。请注意，如果需要对多个属性进行布尔型OR运算，需要使用下一节里介绍的`$or`操作符。

`$in`经常被用于ID列表，本章之前有一个例子，使用`$in`来返回所有购买过特定产品的用户。

① 请注意，数字类型（整型、长整型和双精度浮点数）对这些查询而言在类型上是等价的。

`$nin`仅在与给定元素都不匹配时才返回该文档。可以用`$nin`来查找所有不是黑色或蓝色的产品：

```
db.products.find('details.color': { $nin: ["black", "blue"] }
```

最后，当搜索键与每个给定元素都匹配时，`$all`才会返回文档。如果想查找所有标记为*gift*和*garden*的产品，`$all`是个不错的选择：

```
db.products.find(tags: { $all: ["gift", "garden"] }
```

当然，这条查询只有在以标签数组的形式保存`tags`属性时才有效，比如下面这样：

```
{ name: "Bird Feeder",
  tags: [ "gift", "birds", "garden" ]
}
```

在使用集合操作符时请牢记`$in`和`$all`能利用索引，但`$nin`不能，所以它需要做集合扫描。如果要用`$nin`，试着和一个能用上索引的查询条件一起使用，最好是换种方式来表示这条查询。举个例子，可以再保存一个属性，其中的内容和`$nin`查询等价。例如，假设经常会查询`{timeframe: {$nin: ['morning', 'afternoon']}}`，这时可以换种更直接的方式`{timeframe: 'evening'}`。

4. 布尔操作符

MongoDB的布尔操作符包括`$ne`、`$not`、`$or`、`$and`和`$exists`。

不等于操作符`$ne`的用法可以想象。在实践中，最好和其他操作符结合使用；否则查询效率可能不高，因为它无法利用索引。例如，可以使用`$ne`查找所有由ACME生产并且没有*gardening*标签的产品：

```
db.products.find('details.manufacturer': 'ACME', tags: {$ne: "gardening"} }
```

`$ne`可以作用于单个值和数组，正如示例所示，可以匹配`tags`数组。

`$ne`匹配特定值以外的值，而`$not`则是对另一个MongoDB操作符或正则表达式查询的结果求反。在使用`$not`前，请记住大多数查询操作符已经有否定形式了（`$in`和`$nin`、`$gt`和`$lte`等），`$not`不该和它们搭配使用。当你所使用的操作符或正则表达式没有否定形式时，才应使用`$not`。例如，如果想查询所有姓氏不是*B*打头的用户，可以这样使用`$not`：

```
db.users.find(last_name: {$not: /^B/} }
```

`$or`表示两个不同键对应的值的逻辑或关系。其中重要的一点是：如果可能的值限定在同一个键里，使用`$in`代替。一般而言，查找所有蓝色或绿色产品的语句是这样的：

```
db.products.find('details.color': {$in: ['blue', 'green']} }
```

但是，查找所有蓝色的或者是由ACME生产的产品，就要用`$or`了：

```
db.products.find({ $or: [{'details.color': 'blue'}, {'details.manufacturer':
    'ACME'}] })
```

`$or`接受一个查询选择器数组，每个选择器的复杂度随意，而且可以包含其他查询操作符[①]。

和`$or`一样，`$and`操作符同样接受一个查询选择器数组。对于包含多个键的查询选择器，MongoDB会对条件进行与运算，因此只有在不能简单地表示AND关系时才应使用`$and`。例如，

① 不包括`$or`。

假设想查询所有标记有*gift*或*holiday*，同时还有*gardening*或*landscaping*的产品。表示该查询的唯一途径是关联两个`$in`查询：

```
db.products.find({$and: [
{tags: {$in: ['gift', 'holiday']}},
{tags: {$in: ['gardening', 'landscaping']}}
                         ]
                         }
                         )
```

本节要讨论的最后一个操作符是`$exists`。该操作符的存在很有必要，因为集合没有一个固定的Schema，所以偶尔需要查询包含特定键的文档。你是否记得我们计划在每个产品的`details`属性里保存特定的字段？举例来说，假设要在`details`属性里保存一个`color`字段。但是，如果只有一部分产品中定义了颜色，可以像下面这样将未定义颜色的产品找出来：

```
db.products.find({'details.color': {$exists: false}})
```

也可以查找定义了颜色的产品：

```
db.products.find({'details.color': {$exists: true}})
```

上面只是检查了存在性，还有另一种检查存在性的方式，两者几乎是等价的：用`null`来匹配属性。可以修改上述查询，第一个查询可以这样表示：

```
db.products.find({'details.color': null})
```

第二个是这样的：

```
db.products.find({'details.color': {$ne: null}})
```

5. 匹配子文档

本书的电子商务数据模型中，有些条目里的键指向一个内嵌对象。产品的`details`属性就是一个很好的例子。以下是一个相关文档的片段，用JSON表示：

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  details: {
      model_num: 4039283402,
      manufacturer: "Acme",
      manufacturer_id: 432,
      color:  "Green"
  }
}
```

可以通过.（点）来分隔相关的键，查询这些对象。举例来说，假设想查找所有由ACME生成的产品，可以这样做：

```
db.products.find({'details.manufacturer_id': 432});
```

此类查询里可以指定任意的深度，假设稍微修改一下表述：

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  details: {
      model_num: 4039283402,
      manufacturer: { name: "Acme",
                      id:   432 },
      color:  "Green"
    }
}
```

可以在查询选择器的键里包含两个点：

```
db.products.find({'details.manufacturer.id': 432});
```

除了匹配单个子文档属性，还可以匹配整个对象。例如，假设正使用MongoDB保存股市价位，为了节省空间，放弃了标准的对象ID，用一个包含股票代码和时间戳的复合键取而代之。文档的表述大致是这样的：[①]

```
{ _id: {sym: 'GOOG', date: 20101005}
  open:  40.23,
  high:  45.50,
  low:   38.81,
  close: 41.22
}
```

接下来可以通过如下_id查询获取GOOG于2010年10月5日的价格汇总：

```
db.ticks.find({_id: {sym: 'GOOG', date: 20101005} });
```

一定要注意，像这样匹配整个对象的查询会执行严格的字节比较，也就是说键的顺序很重要。下面的查询与其并不等价，不会匹配到示例文档：

```
db.ticks.find({_id: {date: 20101005, sym: 'GOOG'} });
```

虽然Shell中输入的JSON文档的键顺序会被保留，但并不是所有语言驱动的文档表述都是如此。例如，Ruby 1.8里的散列并不会保留顺序，要在Ruby 1.8中保留键顺序，必须使用BSON::OrderedHash类：

```
doc = BSON::OrderedHash.new
doc['sym']  = 'GOOG'
doc['date'] = 20101005
@ticks.find(doc)
```

一定要检查正使用的语言是否支持有序字典；如果不支持的话，该语言的MongoDB驱动会提供一个有序的替代品。

6. 数组

数组使得文档模型更加强大，如你所见，数组可以用来存储字符串列表、对象ID列表，甚至是其他文档的列表。数组能带来更丰富、更易理解的文档；按照常理，MongoDB能轻松地查询

① 在潜在的高吞吐量场景下，我们希望尽可能地限制文档大小。可以使用较短的键名部分实现该目的，比如用*o*代替*open*。

并索引数组类型。事实也是如此：最简单的数组查询就和其他文档类型的查询一样。仍以产品标签为例，用简单的字符串数组来表示标签：

```
{ _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  tags: ["tools", "equipment", "soil"] }
```

查询带有*soil*标签的产品很简单，使用的语法就和查询单个文档值时一样：

```
db.products.find({tags: "soil"})
```

重要的是，这条查询能利用`tags`字段上的索引。如果在该字段上构建了索引，并且`explain()`该查询，可以看到使用了B树游标：

```
db.products.ensureIndex({tags: 1})
db.products.find({tags: "soil"}).explain()
```

在需要对数组查询拥有更多掌控时，可以使用点符号来查询数组特定位置上的值。下面是如何对之前的查询进行限制，只查询产品的第一个标签：

```
db.products.find({'tags.0': "soil"})
```

如此查询标签可能意义不大，但假设正在处理用户地址，可以用子文档数组来表示地址：

```
{ _id:        ObjectId("4c4b1476238d3b4dd5000001")
  username: "kbanker",

  addresses: [
    {name:     "home",
     street:  "588 5th Street",
     city:     "Brooklyn",
     state:    "NY",
     zip:      11215},

    {name:    "work",
     street: "1 E. 23rd Street",
     city:    "New York",
     state    "NY",
     zip      10010},
  ]
}
```

我们可以规定数组的第0个元素始终是用户的首选送货地址。因此，要找到所有首选送货地址在纽约的用户，可以指定第0个位置，并用点来明确`state`字段：

```
db.users.find({'addresses.0.state': "NY"})
```

我们还可以忽略位置，直接指定字段。如果列表中的任意地址在纽约范围内，下面的查询就会返回用户文档：

```
db.users.find({'addresses.state': "NY"})
```

与之前一样，我们希望为带点的字段加上索引：

```
db.users.ensureIndex({'addresses.state': 1})
```

请注意，无论字段是指向子文档，还是子文档数组，都使用相同的点符号。点符号很强大，而且这种一致性很可靠。但在查询子对象数组中的多个属性时会带来歧义，例如假设想获取所有家庭地址在纽约的用户列表，该如何表示这条查询呢？

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

上述查询的问题在于所引用的字段并不局限于单个地址；换言之，只要有一个地址被设置为"home"，一个地址是在纽约，这条查询就能匹配上了，但我们希望将两个属性都应用到同一个地址上。幸好有一个针对这种情况的查询操作符，要将多个条件限制在同一个子文档上，可以使用$elemMatch操作符，可以这样进行查询：

```
db.users.find({addresses: {$elemMatch: {name: 'home', state: 'NY'}}})
```

从逻辑上来看，只有在需要匹配子文档中的多个属性时才会使用$elemMatch。

唯一还没讨论的数组操作符是$size，该操作符能让我们根据数组大小进行查询。例如，假设希望找出所有带三个地址的用户，可以这样使用$size操作符：

```
db.users.find({addresses: {$size: 3}})
```

在本书编写时，$size操作符是不使用索引的，而且仅限于精确匹配（不能指定数组大小范围）[①]。因此，如果需要基于数组的大小进行查询，应该将大小缓存在文档的属性中，当数组变化时手动更新该值。举例来说，可以考虑为用户文档添加一个address_length字段，并为该字段添加索引，随后再发起范围查询和精确查询。

7. JavaScript

如果目前为止的工具都无法表示你的查询，那就可能需要写一些JavaScript了。我们可以使用特殊的$where操作符，向任意查询中传入一个JavaScript表达式。在JavaScript上下文里，关键字this指向当前文档，让我们来看一个例子：

```
db.reviews.find({$where: "function() { return this.helpful_votes > 3; }"}})
```

该查询还有一个简化形式：

```
db.reviews.find({$where: "this.helpful_votes > 3"}})
```

这个查询能正常使用，但你永远也不会想去使用它，因为可以用标准查询语言轻松表示该查询。问题是JavaScript表达式无法使用索引，由于必须在JavaScript解释器上下文中运算，还带来了额外的大量开销。出于这些原因，应该只在无法通过标准查询语言表示查询时才使用JavaScript查询。如果确实有需要，请尝试为JavaScript表达式带上至少一个标准查询操作符。标准查询操作符可以缩小结果集，减少必须加载到JS上下文里的文档。让我们看个简单的例子，看看为什么需要这么做。

假设为每个用户都计算了一个评分可靠性因子，这是一个整数，与用户的评分相乘之后可以得到一个更标准化的评分。假设后续想查询某个特定用户的评论，并且只返回标准化评分大于3的记录。查询语句是这样的：

① 关于这个问题，更新内容参见https://jira.mongodb.org/browse/SERVER-478。

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            $where: "(this.rating * .92) > 3"})
```

这条查询满足了之前的两条建议：在user_id字段上使用了标准查询，这个字段一般是有索引的；在超出标准查询语言能力的情况下使用了JavaScript表达式。

除了要识别出额外的性能开销，还要意识到JavaScript注入攻击的可能性。当用户可以直接向JavaScript查询中输入代码时就有可能发生注入攻击。虽然用户无法通过这种方式修改或删除数据，但却能获取敏感数据。Ruby中的不安全JavaScript查询可能是这个样子的：

```
@users.find({$where => "this.#{attribute} == #{value}"})
```

假定用户能控制attribute和value的值，他就能以任意属性对来查询集合。虽然这不是最坏情况的入侵，但还是应该尽量避免它。

8. 正则表达式

本章开篇的地方，我们看到在查询中有使用正则表达式，在那个例子里，我演示了前缀表达式/^Ba/，用它来查找以*Ba*开头的姓氏，并且指出这条查询能用上索引。实际上，我们还能使用更多的正则表达式。MongoDB编译时用了PCRE（http://mng.bz/hxmh），它支持大量的正则表达式。

除了之前提到的前缀查询，正则表达式都用不上索引。因此，我建议在使用时和JavaScript表达式一样，结合至少一个其他查询项。下面的例子中，将查询指定用户的包含*best*或*worst*文字的评论。请注意，这里使用了正则表达式标记i[①]来表示忽略大小写：

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            text: /best|worst/i })
```

如果所使用的语言拥有原生的正则表达式类型，我们也可以使用原生的正则表达式对象执行查询。在Ruby中相同的查询语句是这样的：

```
@reviews.find({:user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
               :text => /best|worst/i })
```

如果我们的环境中不支持原生的正则表达式类型，可以使用特殊的$regex和$options操作符。Shell中通过这些操作符可以这样来表示上述查询：

```
db.reviews.find({user_id: ObjectId("4c4b1476238d3b4dd5000001"),
                 text: {$regex: "best|worst", $options: "i" })
```

9. 其他查询操作符

还有两个查询操作符难以归类，所以单独进行讨论。第一个是$mod，允许查询匹配指定取模操作的文档。举例来说，可以通过下列查询找出所有小计能被3整除的订单：

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

我们看到$mod操作符接受两个值组成的数组，第一个值是除数，第二个值是期望的余数。因此，可以这样来理解该查询：找出所有小计除以3后余0的文档。这个例子是故意做出来的，但它能体现出背后的思想。如果要使用$mod操作符，请牢记它无法使用索引。

① 使用了忽略大小写的选项就无法在查询中使用索引，就算是在前缀匹配时也是如此。

第二个操作符是$type，根据BSON类型来匹配值。我不建议为一个集合的同一个字段保存多种类型，但是如果发生这样的情况，可以用这个操作符来检查类型。我最近发现某个用户的_id查询总是匹配不上数据，而实际上不应该发生这样的情况，这时$type操作符就能派上用场了。问题的原因是他既将ID保存为字符串，又将其保存为对象ID，它们的BSON类型分别是2和7，对于新用户而言，很容易就会忽略两者的区别。

要修正这个问题，首先要找出所有以字符串形式保存ID的文档。使用$type操作符就可以了：

```
db.users.find({_id: {$type: 2}})
```

5.2.2　查询选项

所有的查询都要有一个查询选择器。就算没有提供，查询本身实际就是由查询选择器定义的。但在发起查询时，有多种查询选项可供选择，它们能进一步约束结果集。本节将介绍这些选项。

1. 投影

在查询结果集的文档中，可以使用投影来选择字段的子集进行返回。当有大文档时就更应该使用投影，这能最小化网络延时和反序列化的开销。通常是用要返回的字段集合来定义投影：

```
db.users.find({}, {username: 1})
```

该查询返回的用户文档只包含两个字段：username和_id。默认情况下，_id字段总是包含在返回结果内。

在某些情况下，你可能还会希望排除特定字段。举例来说，本书的用户文档中包含送货地址和支付方式，但通常并不需要这些信息，为了将其排除掉，可以在投影中添加这些字段，并将其值设置为0：

```
db.users.find({}, {addresses: 0, payment_methods: 0})
```

除了包含和排除字段，还能返回保存在数组里的某个范围内的值。例如，我们可能想在产品文档中保存产品评论，同时还希望能对那些评论进行分页，为此可以使用$slice操作符。要返回头12篇评论或者倒数5篇评论，可以像这样使用$slice：

```
db.products.find({}, {reviews: {$slice: 12}})
db.products.find({}, {reviews: {$slice: -5}})
```

$slice还能接受两个元素的数组，分别表示跳过的元素数和返回元素个数限制。下面演示如何跳过头24篇评论，并限制仅返回12篇评论：

```
db.products.find({}, {reviews: {$slice: [24, 12]}})
```

最后，注意$slice并不会阻止返回其他字段。如果希望限制文档中的其他字段，必须显式地进行控制。例如，修改上述查询，仅返回评论及其评分：

```
db.products.find({}, {reviews: {$slice: [24, 12]}, 'reviews.rating': 1})
```

2. 排序

所有的查询结果都能按照一个或多个字段进行升序或降序排列。例如，根据评分对评论做排

序，从高到低降序排列：

```
db.reviews.find({}).sort({rating: -1})
```

显然，先根据有用程度排序，随后再是评分，这样的排序可能更有价值：

```
db.reviews.find({}).sort({helpful_votes:-1, rating: -1})
```

在类似的组合排序里，顺序至关重要。正如书中其他地方所说的，Shell中键入的JSON是有顺序的。因为Ruby的散列是无序的，所以可以用数组的数组来指定排序顺序，数组是有序的：

```
@reviews.find({}).sort([['helpful_votes', -1], [rating, -1]])
```

在MongoDB中指定排序非常简单，但书中其他章节里讨论到的两个主题对理解排序来说必不可少。其一，了解如何使用`$natural`操作符根据插入顺序进行排序，这是在第4章里讨论的。其二，这点就更有关系了，即了解如何保证排序能有效利用到索引，第8章里会讨论这个主题。如果正在大量使用排序，可以先阅读第8章。

3. skip与limit

`skip`与`limit`的语义很容易理解，这两个查询选项的作用总能满足预期。

但在向`skip`传递很大的值（比如大于10 000的值）时需要注意，因为执行这种查询要扫描和`skip`值等量的文档。例如，假设正根据日期降序对100万个文档进行分页，每页10条结果。这意味着显示第50 000页的查询要跳过500 000个文档，这样做的效率太低了。更好的策略是省略`skip`，添加一个范围条件，指明下一结果集从何处开始。如此一来，这条查询：

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

就变成了：

```
db.docs.find({date: {$gt: previous_page_date}}).limit(10).sort({date: -1})
```

第二条查询扫描的文档远少于第一条。唯一的问题是如果每个文档的日期不唯一，相同的文档可能会显示多次。有很多应对这种情况的策略，寻找解决方案的任务就留给读者了。

5.3 聚合指令

我们已经看过MongoDB的聚合命令`count`的例子了（`count`被用于分页）。大多数数据库都提供了`count`和其他很多内置的聚合函数，用于计算总和、平均数、方差等。这些特性都在MongoDB的规划之中，但在实现前，我们可以使用`group`与`map-reduce`编写脚本实现各种聚合函数，从简单的求和到计算标准差。

5.3.1 根据用户对评论进行分组

通常人们都想知道哪些用户提供了最有价值的评论。既然应用程序允许用户为评论投票，那么从技术上讲，就能计算出某个用户所有评论的总得票数，以及该用户每篇评论的平均得票数。虽然可以查询所有评论并执行一些基本的客户端处理来获取这些统计信息，但还可以使用

MonogoDB的group命令从服务器获取结果。

group最少需要三个参数。第一个参数是key，定义如何对数据进行分组。本例中，我们希望结果根据用户分组，因此分组的键是user_id。第二个参数是一个对结果集做聚合的JavaScript函数，叫reduce函数。第三个分组参数是reduce函数的初始文档。

实际并没有听上去那么复杂。让我们仔细看看将用到的初始文档，以及相应的reduce函数：

```
initial = {review: 0, votes: 0};

reduce  = function(doc, aggregator) {
  aggregator.reviews += 1.0;
  aggregator.votes   += doc.votes;
}
```

我们看到初始化文档为每个分组键定义了一些值，换言之，每次运行group，我们都希望针对每个user_id得到一个结果集，其中包含写过的评论总数，以及所有那些评论的总得票数。生成这些总和的工作是由reduce函数完成的。假设我写了五条评论，也就是说有五个评论文档中标记有我的用户ID，这五个文档都会被分别传递给reduce函数，作为doc参数。一开始aggregator的值是initial文档，后续每处理一个文档就会往aggregator里添加值。

下面展示如何在JavaScript Shell中执行group命令。

代码清单5-1　使用MongoDB的group命令

```
results = db.reviews.group({
  key:      {user_id: true},
  initial:  {reviews: 0, votes: 0.0},
  reduce:   function(doc, aggregator) {
               aggregator.reviews += 1;
               aggregator.votes   += doc.votes;
            }
  finalize: function(doc) {
               doc.average = doc.votes / doc.reviews;
            }
})
```

请注意，此处向group传递了一个额外的参数。我们希望获得每篇评论的平均得票数，但在计算出总的评论得票数和评论总数之前，无法得到该值。这就是使用终结器（finalizer）的原因，它是一个JavaScript函数，在group命令返回前应用于每个分组结果上。本例中，我们使用终结器计算每篇评论的平均得票数。

下面是针对示例数据集运行以上聚合的结果。

代码清单5-2　group命令的结果

```
[
  {user_id: ObjectId("4d00065860c53a481aeab608"),
   votes: 25.0,
   reviews: 7,
   average: 3.57
  },

  {user_id: ObjectId("4d00065860c53a481aeab608"),
```

```
    votes: 25.0,
    reviews: 7,
    average: 3.57
  }
]
```

本章结尾处我们还会谈到group命令，包括它所有的选项和特质。

5.3.2 根据地域对订单应用MapReduce

我们可以把MongoDB的map-reduce当做更灵活的group。有了map-reduce，可以更细粒度地控制分组键，还有大量输出选项可用，包括将结果存储在新的集合里，以便后续能够更方便地获取那些数据。让我们通过一个例子来了解两者在实践中的不同。

我们有时希望生成一些销售汇总，可以以此为例。每个月销售量有多少？过去一年里每个月的销售额有多少？通过map-reduce可以很方便地回答这些问题。正如map-reduce的名字所暗示的，第一步就是编写一个映射函数，应用于集合里的每个文档，在此过程中实现两个目的：定义分组所用的键，整理计算所需的所有数据。要实际了解这个过程，可以仔细查看以下函数：

```
map = function() {
    var shipping_month = this.purchase_date.getMonth() +
      '-' + this.purchase_data.getFullYear();

    var items = 0;
    this.line_items.forEach(function(item) {
    items += item.quantity;
    });

    emit(shipping_month, {order_total: this.sub_total, items_total: items});
  }
```

首先，需要知道变量this总是指向正在迭代的文档。在函数的第一行里，我们获取了一个表示订单创建月份的整数[①]。随后调用了emit()，这是每个映射函数必须要调用的特殊方法。emit()的第一个参数是分组依据的键，第二个参数通常是包含要执行reduce的值的文档。本例中，我们要根据月份分组，对每个订单的小计和明细项数量做统计。看了与之对应的reduce函数之后，一切就再明白不过了：

```
reduce = function(key, values) {
    var tmpTotal = 0;
    var tmpItems = 0;

    values.forEach(function(doc) {
      tmpTotal += doc.order_total;
      tmpItems += doc.items_total;
    });
    return ( {total: tmpTotal, items_total : tmpItems} );
  }
```

① 因为JavaScript的月份是从0开始的，所有该值的范围是0~11。我们需要在此基础上加1，这样的月份表述更加直观。后面加了-和年份，因此整个键看起来是这样的：1-2011、2-2011，以此类推。

reduce函数接受一个键和一个包含一个或多个值的数组。编写reduce函数时要确保那些值按照既定的方式进行聚合，并且能返回单个值。因为map-reduce的迭代本质，reduce可能被执行多次，而编写代码时必须把这种情况也考虑在内。在实践中，这就意味着对于一个映射函数给出的值而言，多次执行reduce函数的返回值必须保证是相同的。仔细想想，你会发现情况就是这样的。

Shell的map-reduce方法要求提供一个映射函数和一个reduce函数作为参数。本例中还增加了另外两个参数。第一个参数是查询过滤器，将聚合操作所涉及的文档限制在2010年之后创建的订单。第二个参数是输出集合的名称。

```
filter = {purchase_date: {$gte: new Date(2010, 0, 1)}}
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'})
```

该操作的结果保存在名为totals的集合之中，我们可以像查询其他集合一样对它进行查询。下面的代码显示了对totals集合的查询结果。_id字段是分组键，其中的内容是年和月；value字段是统计出的汇总信息。

代码清单5-3　查询map-reduce的输出集合

```
> db.totals.find()
{ _id: "1-2011", value: { total: 32002300, items: 59 }}
{ _id: "2-2011", value: { total: 45439500, items: 71 }}
{ _id: "3-2011", value: { total: 54322300, items: 98 }}
{ _id: "4-2011", value: { total: 75534200, items: 115 }}
{ _id: "5-2011", value: { total: 81232100, items: 121 }}
```

本节的示例从实践出发让我们对MongoDB的聚合能力有了感性的了解，下一节的内容会涵盖它的大部分细节内容。

5.4　详解聚合

本节我将对MongoDB的聚合函数做详细说明。

5.4.1　max()与min()

通常总是需要找到给定集合里的最大和最小值。使用SQL的数据库提供了min()和max()函数，但MongoDB没有这样的函数，我们必须自己实现。要找到某个字段中的最大值，可以按照该字段降序排序，并限制结果集为一个文档；按照相反顺序排序就能取到对应的最小值。例如，如果希望找到投票数最多的评论，查询需要对投票的字段进行排序，限制返回一个文档：

```
db.reviews.find({}).sort({helpful_votes: -1}).limit(1)
```

返回文档中的helpful_votes字段包含了该字段中的最大值。要获取最小值，只要逆序排列就行了：

```
db.reviews.find({}).sort({helpful_votes: 1}).limit(1)
```

如果要在生产环境中发起查询，helpful_votes字段最好能有一个索引。如果想获得特定

产品里投票数最多的评论，则需要一个product_id和helpful_votes的复合索引。如果不清楚这么做的原因，可以阅读第7章。

5.4.2 distinct

MongoDB的distinct命令是获取特定字段中不同值列表的最简单工具。该命令既适用于单键，也适用于数组键。distinct默认覆盖整个集合，但也可以通过查询选择器进行约束。

可以像下面这样使用distinct获取产品集合里所有唯一标签的列表：

```
db.products.distinct("tags")
```

这很简单。如果希望操作products集合的一个子集，可以传入一个查询选择器作为第二个参数。这里的查询将不同的标签值限定到Gardening Tools分类里的产品：

```
db.products.distinct("tags",
          {category_id: ObjectId("6a5b1476238d3b4dd5000048")})
```

聚合命令限制

在实用性方面，distinct和group有一个很大的限制：它们返回的结果集不能超过16 MB。16 MB的限制并不是这些命令本身所强加的阈值，这是所有的初始查询结果集大小。distinct和group是以命令的方式实现的，也就是对特殊的$cmd集合的查询，它们赖以生存的查询则受制于该限制。如果distinct或group处理不了你的聚合结果集，那么就只能使用map-reduce代替了，它的结果可以保存在集合中而非内联（inline）返回。

5.4.3 group

group和distinct一样，也是数据库命令，因此它的结果集也受制于同样的16 MB响应限制。而且，为了减少内存消耗，group不会处理多于10 000个唯一键。如果聚合操作在此范围内，group是个不错的选择，因为通常情况下它会比map-reduce快。

我们已经看过根据用户对评论分组的例子了，那个示例只能算“半个”。让我们快速回顾一下传递给group的选项。

- key，描述分组字段的文档。举例来说，要根据category_id分组，可以将{category_id: true}作为键。此处还可以使用复合键，比如，若想根据user_id和rating对一系列帖子做分组，键看起来是这样的：{user_id: true, rating: true}。除非使用keyf，否则key选项是必需的。
- keyf，这是一个JavaScript函数，应用于文档之上，为该文档生成一个键，当用于分组的键需要计算时，这个函数非常有用。举例来说，如果想根据每个文档创建时是周几来对结果集进行分组，但又不实际存储该值，就可以用键函数来生成这个键：

```
function(doc) {
  return {day: doc.created_at.getDay();
}
```

这个函数会生成类似{day: 1}这样的键。请注意，如果没有指定标准的key，那么keyf是必需的。

- initial，作为聚合结果初始值的文档。reduce函数第一次运行时，该初始文档会作为聚合器的第一个值，通常会包含所有要聚合的键。举例来说，如果正在为每个分组项计算总投票数和总文档数，那么初始文档看起来是这样的：{vote_sum: 0.0, doc_count: 0}。

请注意，该参数是必需的。

- reduce，用于执行聚合的JavaScript函数。该函数接受两个参数：正被迭代的当前文档和用于存储聚合结果的聚合器文档。聚合器的初始值就是初始文档。下面是一个聚合投票和文档总数的reduce函数示例：

```
function(doc, aggregator) {
  aggregator.doc_count += 1;
  aggregator.vote_sum  += doc.vote_count;
}
```

请注意，reduce函数并不返回任何内容，它只不过是修改聚合器对象。reduce函数也是必需的。

- cond，过滤要聚合文档的查询选择器。如果不希望分组操作处理整个集合，就必须提供一个查询选择器。例如，假设只想聚合那些拥有五个以上投票的文档，可以提供以下查询选择器：{vote_count: {$gt: 5}}。
- finalize，在返回结果集之前应用于每个结果文档的JavaScript函数。该函数支持对分组操作的结果进行后置处理。我们通常会用它计算平均值，在分组结果的现有值之外，再加另一个值来保存平均值：

```
function(doc) {
  doc.average = doc.vote_count / doc.doc_count;
}
```

诚然，group有这么多选项，上手比较麻烦。但是，稍加实践之后，你会很快习惯的。

5.4.4　map-reduce

既然group和map-reduce提供了类似的功能，你可能会想MongoDB为什么要同时对它们提供支持呢？其实，在添加map-reduce之前，group是MongoDB唯一的聚合器，map-reduce是后来出于一些原因加入的。首先，MapReduce风格的操作正在成为主流，而且将这种思考方式融入产品之中看起来是很明智的。[①]其次，也是更实际的原因：对大数据集进行迭代，尤其是在分片配置中，需要有分布式的聚合器，而MapReduce（范式）恰恰提供所需的内容。

map-reduce包含很多选项。此处详细对这些选项做了说明。

① 很多开发者是在谷歌那篇著名的关于分布式计算的论文（http://labs.google.com/papers/mapreduce.html）里初次看到MapReduce的。其中的思想后来成了Hadoop的基础，而Hadoop是一个使用分布式MapReduce处理大数据集的开源框架。之后MapReduce的思想得到了广泛传播，例如CouchDB就用MapReduce的范式来声明索引。

- map，应用于每个文档之上的JavaScript函数。该函数必须调用emit()来选择要聚合的键和值。在函数上下文中，this的值指向当前文档。例如，假设想根据用户ID对结果分组，计算出总投票数和总文档数，映射函数应该是这样的：

```
function() {
  emit(this.user_id, {vote_sum: this.vote_count, doc_sum: 1});
}
```

- reduce，一个JavaScript函数，接受一个键和一个值列表。该函数对返回值的结构有严格要求，必须总是与values数组所提供的结构一致。reduce函数通常会迭代一个值的列表，在此过程中对其进行聚合。回到我们的示例，以下展示如何处理映射函数输出的内容：

```
function(key, values) {
  var vote_sum = 0;
  var doc_sum  = 0;

  values.forEach(function(value) {
    vote_sum += value.vote_sum;
    doc_sum  += value.doc_sum;

  });
  return {vote_sum: vote_sum, doc_sum: doc_sum};
}
```

  请注意，通常在聚合过程中不会用到key参数的值。

- query，用于过滤映射处理的集合的查询选择器。该参数的作用与group的cond参数相同。
- sort，对于查询的排序。与limit选项搭配使用时非常有用，这样就可以对1000个最近创建的文档运行map-reduce。
- limit，一个整数，指定了查询和排序的条数。
- out，该参数决定了如何返回输出内容。要将所有输出作为命令本身的结果，传入{inline: 1}。请注意，这仅适用于结果集符合16 MB返回限制的情况。

另一个选择是将结果放到一个输出集合里。此时，out的值必须是一个字符串，标明用于保存结果的集合的名称。

将结果保存到输出集合时有一个问题：如果最近运行过类似的map-reduce，那么可能会覆盖现有数据。因此，还有两个集合输出选项：一个用于合并结果和老数据，另一个对数据进行reduce处理。在合并的场景中，使用{merge: "collectionName"}，新结果会覆盖拥有相同键的现有项。如果使用{reduce: "collectionName"}，会调用reduce函数根据新值来处理现有键的值。尤其是在执行要反复运行的MapReduce任务时，希望把新数据整合到已有的聚合之中，reduce格外有用。在对集合执行新的MapReduce任务时，只需简单添加一个查询选择器来限制聚合所需的数据集。

- finalize，一个JavaScript函数，在reduce阶段完成后会应用于每个返回的文档上。
- scope，该文档指定了map、reduce和finalize函数可全局访问的变量的值。
- verbose，一个布尔值，为true时，在命令返回文档中会包含对map-reduce任务执行时间的统计信息。

在考虑使用MongoDB的`map-reduce`和`group`时，还有一个重要的限制需要引起注意：速度。对于大的数据集，这些聚合函数通常执行起来满足不了用户对速度的需求。这几乎都要归咎于MongoDB的JavaScript引擎。一个单线程解释（非编译）运行的JavaScript引擎是很难实现高性能的。

但也不要沮丧，`map-reduce`和`group`被广泛使用于很多场景之中，并能充分胜任这些任务。对于那些还不适用的场景，则有其他方案，并有望在未来提供支持。其他方案是指在别处执行聚合，拥有大数据集的用户已经在Hadoop集群上成功处理过数据。未来有望加入新的聚合函数，它们使用编译的多线程代码。这些功能计划于MongoDB v2.0之后的某个时间发布，你可以关注https://jira.mongodb.org/browse/SERVER-447。

5.5　小结

查询与聚合是MongoDB接口中的重要部分。因此在阅读了本章的内容之后，强烈建议你试用查询与聚合机制。如果不确定如何利用查询操作的某些特定组合，Shell总是测试的最佳场所。

从现在起我们会一直使用MongoDB查询，下一章里就是如此。下一章中我们会进行文档更新，因为查询在大多数更新中都起关键作用，所以你会对本章所介绍的查询语言有更多了解。

第 6 章

更新、原子操作与删除

本章内容

- ❑ 更新文档
- ❑ 原子性地处理文档
- ❑ 分类层级与库存管理

更新就是向现有文档写入新的内容，想要高效地进行更新操作，需要彻底了解各种可用的文档结构类型和MongoDB提供的查询表达式。通过上两章了解了电子商务数据模型之后，你应该对Schema的设计和查询方式有所感悟了。在学习更新时，这些知识都能派上用场。

我们会深入探讨为什么采用这样一种去正规化的方式来建模分类层级，以及MongoDB的更新操作是如何让这种结构变得合理的。我们还会探讨库存管理，在此过程中解决一些棘手的并发问题。你将认识很多新的更新操作符，了解一些能充分利用更新操作原子性的技巧，体会`findAndModify`命令的强大之处。在众多示例之后，会有一节专门描述各个更新操作符的细节。我还会介绍一些与并发和优化相关的注意事项，最后简明扼要地概述一下MongoDB中如何删除数据。

到本章结束时，你将认识MongoDB的全部CRUD操作，能顺利地设计应用程序，充分利用MongoDB的接口与数据模型了。

6.1 文档更新入门

如果需要在MongoDB中更新文档，有两种方式，既可以整个替换文档，也可以结合一些更新操作符修改文档中的特定字段。为了替更详细的例子做些铺垫，本章会从一个简单的演示开始，示范这两种做法。随后，我会解释哪种方式更好。

先让我们回忆一下用户示例文档。该文档包含用户的姓名、电子邮件地址和送货地址。毫无疑问，我们时不时会修改一下电子邮件地址，因此就从它开始吧。要完整替换文档，先查询该文档，随后在客户端进行修改，最后用修改后的文档发起更新。以下是对应的Ruby代码：

```
user_id = BSON::ObjectId("4c4b1476238d3b4dd5000001")
doc     = @users.find_one({:_id => user_id})

doc['email'] = 'mongodb-user@10gen.com'
@users.update({:_id => user_id}, doc, :safe => true)
```

有了用户的_id，可以先查询文档。接下来在本地进行修改，这里是修改email属性。随后将修改过的文档传给update方法。最后一行的意思是"找到users集合中指定_id的文档，用我提供的文档替换它"。

以上展示了如何通过替换进行修改，现在让我们看看如何通过操作符进行修改：

```
@users.update({:_id => user_id},
  {'$set' => {:email => 'mongodb-user@10gen.com'}},
  :safe => true)
```

本例中使用$set在一个服务器请求里修改了电子邮件地址，这是多个特殊更新操作符中的一个。这里的更新请求更有针对性：找到指定用户文档，将其email字段设置为mongodb-user@10gen.com。

其他示例又会如何？这次，我们向用户的地址列表中添加其他送货地址，下面展示如何通过文档替换实现该操作：

```
doc = @users.find_one({:_id => user_id})

new_address = {
       :name   => "work",
       :street => "17 W. 18th St.",
       :city   => "New York",
       :state  => "NY",
       :zip    => 10011
}
doc['shipping_addresses'].append(new_address)
@users.update({:_id => user_id}, doc)
```

更有针对性的做法是这样的：

```
@users.update({:_id => user_id},
  {'$push' => {:shipping_addresses =>
    {:name => "work",
   :street => "17 W. 18th St.",
   :city   => "New York",
    :state => "NY",
   :zip    => 10011
      }
    }
  })
```

替换的方法与之前类似，从服务器获取用户文档，进行修改，随后发回服务器。此处的更新语句和更新电子邮件地址时的一样。相比之下，针对性更新里使用了不同的操作符$push，将新地址推送到现有的shipping_addresses数组里。

既然已经看过了几个实际的更新，请思考一下，已经有了一种方法后为什么还要用另一种呢？你觉得哪种方式更直观，哪种方式的性能会更好？

替换更新是种更通用的方式。假设应用程序显示了一个用于更新用户信息的HTML表单，使用替换更新时，从表单提交的数据一经校验就能直接传入MongoDB；无论修改了哪个用户属性，执行更新的代码都是一样的。举例来说，如果你打算构建一个MongoDB对象映射器，需要通用

的更新，那么替换更新可能更适合作为默认值。[1]

针对性更新通常性能会更好。首先，不需要在开始时到服务器上获取要修改的文档。其次，指定更新内容的文档一般都很小。如果是通过替换进行更新，文档的平均大小是100 KB，那么每次更新都要向服务器发送100 KB内容！相比之下，上个例子里，无论要修改的文档有多大，每个使用`$set`和`$push`来指定更新的文档都小于100字节。为此，经常使用针对性更新就意味着节省序列化和传输数据的时间。

此外，针对性操作允许原子性地更新文档。举例来说，如果需要增加计数器值，通过替换进行更新就很不理想；唯一能对它们进行原子性更新的方法就是采用某类乐观锁。在针对性更新中，可以使用`$inc`原子性地修改计数器。也就是说，就算有大量的并发更新，每次执行`$inc`都是相互隔离的，要么成功，要么失败。[2]

乐观锁

乐观锁即乐观并发控制，这项技术保证在无需锁定记录的情况下对其进行彻底更新。要理解它，最简单的方法是想象一个wiki，有多个用户可以同时编辑一个wiki页面，但你肯定不希望用户编辑并更新一个过期的页面，这时可以使用乐观锁协议。当用户试图保存他们的变更时，会在更新操作中包含一个时间戳，如果该值比这个页面最近保存的版本旧，那么不能让用户进行更新；但如果没人修改过这个页面，则允许更新。该策略允许多个用户同时编辑一个页面，比另一种要求每个用户在编辑任意页面时获得一个锁的并发策略要好很多。

知道了可用的更新种类之后，你就能理解下一节里我将介绍的策略了。下一节中，我们会回到电子商务数据模型，回答一些与在生产环境中操作数据相关的、更困难的问题。

6.2　电子商务数据模型中的更新

在股票的例子里更新MongoDB文档中的这个或那个属性是很容易的。但生产环境中的数据模型和真实的应用程序中，会出现不少困难，对指定属性的更新可能不再是简单的一行语句。在后续的内容里，我会使用上两章里出现的电子商务数据模型作为示例，演示在生产环境中的电子商务网站里可能看到的更新。其中某些更新很直观，而另一些则不那么直观。但总的来说，我们会对第4章中设计的Schema有更深入的认识，对MongoDB更新语言的特性和限制有更进一步的理解。

---

① 大多数MongoDB对象映射器都采用这种策略，原因也很简单。如果用户可以建模任意复杂度的实体，那么发起替换更新比计算特殊更新操作符的理想组合要方便得多。

② MongoDB文档中使用*原子更新*（atomic update）这个词来表示我所说的*针对性更新*（targeted update）。这个新术语意在突出原子这个词。实际上，所有发往核心服务器的更新都是原子性的，以文档为单位进行隔离。说更新操作符是原子性的是因为它们能在不用先查询的情况下更新文档。

6.2.1　产品与分类

本节你将看到一些实际的针对性更新的例子，首先了解如何计算产品平均评分，随后是更复杂的维护分类层级的任务。

1. 产品平均评分

产品可以运用很多更新策略。假设管理员有一个界面可用于编辑产品信息，最简单的更新涉及获取当前产品文档，将其与用户编辑的文档进行合并，执行一次文档替换。有时候，可能只需更新几个值，显然这时针对性更新是更好的选择。计算产品平均评分就是这种情况。因为用户会基于平均评分对产品列表排序，可以将该评分保存在产品文档中，在添加或删除评论时进行更新。

执行此类更新的方式很多，下面就是其中之一：

```
average = 0.0
count   = 0
total   = 0
cursor = @reviews.find({:product_id => product_id}, :fields => ["rating"])
while cursor.has_next? && review = cursor.next()
  total += review['rating']
  count += 1
end

average = total / count

@products.update({:_id => BSON::ObjectId("4c4b1476238d3b4dd5003981")},
   {'$set' => {:total_reviews => count, :average_review => average}})
```

这段代码聚合并处理了每条产品评论中的rating字段，然后计算了平均值。实际上，我们迭代了每个评分，借此计算产品的总评分，这节省了一次额外的count函数调用。有了评论的总条数和平均评分之后，在代码中使用$set执行一次针对性更新。

关注性能的用户可能会尽量避免每次更新时重新聚合所有产品评论这种做法。此处提供的方法虽然保守，但在大多数情况下还是可以接受的。也会有其他策略，举例来说，可以在产品文档中保存额外的字段来缓存评论的总评分。在插入一条新评论后，查询产品以获得当前评论总数和总评分，随后计算平均值，用如下选择器发起一次更新：

```
{'$set' => {:average_review => average, :ratings_total => total},
 '$inc' => {:total_reviews => 1}})
```

只有使用典型数据对系统进行评测之后才能确定哪种方式更好。但这个例子恰恰说明了MongoDB通常提供多种可选方法，应用程序的需求可用于帮助确定哪种方法是最好的。

2. 分类层级

在很多数据库中都没有简单的方法来表示分类层级，虽然文档结构对此有所帮助，但MongoDB里的情况也差不多。文档可以针对读取进行优化，因为每个分类都能包含其祖先的列表。唯一麻烦的要求是始终保持最新的祖先列表。让我们来看一个例子。

首先需要一个通用的方法更新任意给定分类的祖先列表。下面是一个可行方案：

```
def generate_ancestors(_id, parent_id)
  ancestor_list = []
  while parent = @categories.find_one(:_id => parent_id) do
    ancestor_list.unshift(parent)
    parent_id = parent['parent_id']
  end

  @categories.update({:_id => _id},
    {"$set" {:ancestors => ancestor_list}})
end
```

该方法回溯了分类层级，连续查询每个节点的parent_id属性，直到根节点（parent_id是nil的节点）为止。总之，它构建了一个有序的祖先列表，保存在ancestor_list数组里。最后，使用$set更新分类的ancestors属性。

既然已经有了基本的构建模块，那就让我们来看看插入新分类的过程吧。假设有一个简单的分类层级，如图6-1所示。

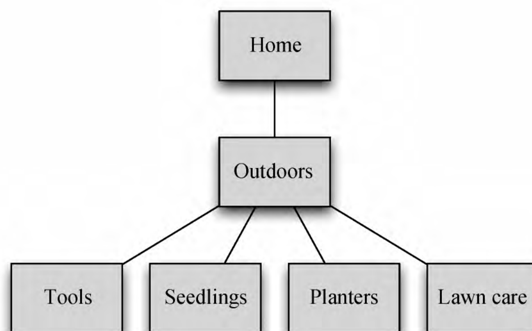


图6-1　初始的分类层级

假设想在Home分类下添加一个名为Gardening的新分类，插入新分类文档后运行方法来生成它的祖先列表：

```
category = {
  :parent_id => parent_id,
  :slug => "gardening",
  :name => "Gardening",
  :description => "All gardening implements, tools, seeds, and soil."
}
gardening_id = @categories.insert(category)
generate_ancestors(gardening_id, parent_id)
```

图6-2显示了更新后的树。

这太简单了。但如果现在想把Outdoors分类放在Gardening下面又会怎么样呢？这就有点复杂了，因为要修改很多分类的祖先列表。可以从把Outdoors的parent_id修改为Gardening的_id开始做起，这还不是很困难：

```
@categories.update({:_id => outdoors_id},
                   {'$set' => {:parent_id => gardening_id}})
```

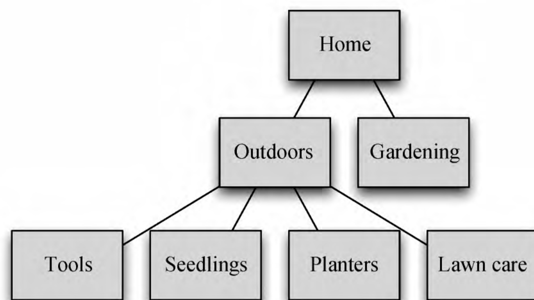


图6-2　添加Gardening分类

因为移动了Outdoors分类，所以其所有后代的祖先列表都无效了。可以查询所有祖先列表里有Outdoors的分类，随后重新生成它们的祖先列表。MongoDB可深入数组进行查询，因而能轻而易举地完成这项工作：

```
@categories.find({'ancestors.id' => outdoors_id}).each do |category|
  generate_ancestors(category['_id'], outdoors_id)
end
```

这就是一个处理分类parent_id属性的更新的方法，图6-3显示了变更后的分类排列方式。

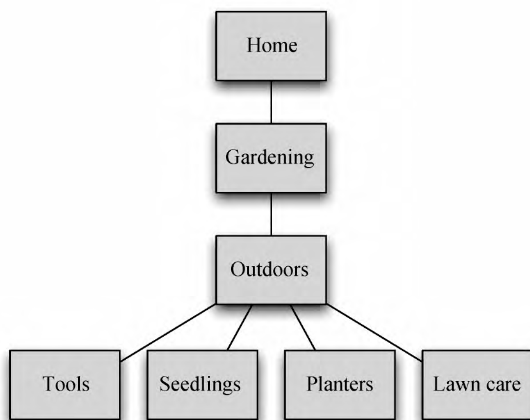


图6-3　最终状态的分类树

要是想要修改分类名称又会怎么样呢？如果将Outdoors分类的名称改为The Great Outdoors，那么还必须修改其他祖先列表中出现Outdoors的分类。这时你会想“看到没？这种情况下去正规化就麻烦了”。但了解到不用重新计算祖先列表就能执行这个更新后，你应该会感觉好多了。方法如下：

```
doc = @categories.find_one({:_id => outdoors_id})
doc['name'] = "The Great Outdoors"
@categories.update({:_id => outdoors_id}, doc)
```

```
@categories.update({'ancestors.id' => outdoors_id},
  {'$set' => {'ancestors.$'=> doc}}, :multi => true)
```

我们先取得了Outdoors文档，在本地修改它的name属性，随后通过替换进行更新，最后再用修改后的Outdoors文档来替换多个祖先列表中的旧文档。我们通过位置操作符和多项更新实现了这个操作。多项更新很容易理解；回忆一下，如果希望修改能作用于所有选择器匹配到的文档，需要指定:multi => true。此处，我们想更新所有祖先列表中有Outdoors的分类。

位置操作符更巧妙一些。假设无从获知Outdoors分类会出现在给定分类祖先列表中的什么地方，此时就需要更新操作符针对任意文档动态定位Outdoors分类在数组中的位置。说到位置操作符，即ancestors.$中的$，代替了查询选择器匹配到的数组下标，这才使这个更新操作成为可能。

因为需要更新数组中单独的子文档，总是会用到位置操作符。总的来说，在要处理子文档数组时，这些更新分类层级的技术都能适用。

6.2.2 评论

评论并不是完全“平等”的，这就是应用程序会允许用户对评论进行投票的原因。投票很简单，它们指出了哪些评论是有用的。我们已经对评论做了建模，其中能缓存总投票数以及投票者ID的列表。评论文档中相关的部分看起来是这样的：

```
{helpful_votes: 3,
voter_ids: [ ObjectId("4c4b1476238d3b4dd5000041"),
             ObjectId("7a4f0376238d3b4dd5000003"),
             ObjectId("92c21476238d3b4dd5000032")
           ]}
```

可以通过针对性更新来记录用户投票。使用$push操作符将投票者的ID添加到列表里，使用$inc操作符来增加总投票数，这两个操作都在同一个更新操作里：

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")},
  {$push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
 $inc: {helpful_votes: 1}
})
```

大多数情况下这个更新没有问题，但我们需要确保仅当正在投票的用户尚未对该评论投过票时才能进行更新。因此要修改此处的查询选择器，只匹配voter_ids数组中不包含要添加的ID的情况。使用$ne查询操作符就能轻松实现了：

```
query_selector = {_id: ObjectId("4c4b1476238d3b4dd5000041"),
  voter_ids: {$ne: ObjectId("4c4b1476238d3b4dd5000001")}}
db.reviews.update(query_selector,
  {$push: {voter_ids: ObjectId("4c4b1476238d3b4dd5000001")},
   $inc : {helpful_votes: 1}
  })
```

这是一个很强大的示例，演示了MongoDB的更新机制以及如何将其用于面向文档的Schema。本例中的投票既是原子操作，又有很高的效率。原子性保证了即使在高并发环境下，也没人能投两次票。高效是因为对投票者身份的判断、更新计数器和投票者列表的操作都是在同一个服务器

请求内完成的。

现在，如果最终确定使用该技术来保存投票信息，请务必保证其他对评论文档的更新都是针对性更新，因为替换更新的方式一定会导致不一致性。想象一下，假设用户通过替换更新来修改评论的内容，先要查询希望修改的文档，在查询评论和更新之间，另一个用户很有可能在为该评论投票。图6-4中就描述了这个事件序列。

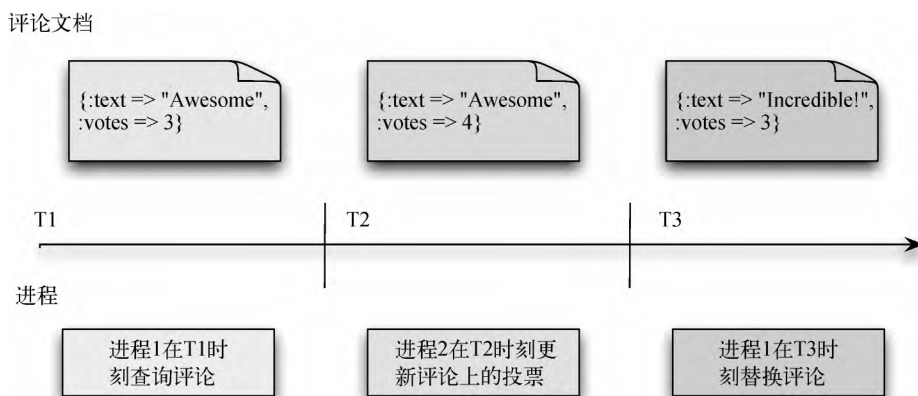


图6-4　通过针对性更新和替换更新并发地修改评论时会丢失数据

很明显，T3时刻的文档替换会覆盖T2时刻发生的投票更新。使用之前描述的乐观锁技术是可以避免这种情况的，但确保本例中所有的更新都是针对性更新似乎更容易一些。

6.2.3　订单

在评论中看到的更新操作的原子性和高效性也能被运用在订单上。接下来，我们会看到如何使用针对性更新实现“添加到购物车”功能（Add to Cart）。这个过程有两步：第一步，构建一个产品文档，用来保存订单条目数组；第二步，发起一次针对性更新，标明这是一次*upsert* ——如果要更新的文档不存在则插入一个新文档的更新操作。（在下一节里我会详细描述upsert的。）如果订单对象不存在，该操作会创建一个新的订单对象，无缝地处理初始化以及后续“添加到购物车”的动作。[①]

我们先构建一个要添加到购物车中的示例文档：

```
cart_item = {
  _id:  ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku:  "9092",

  name: "Extra Large Wheel Barrow",
```

① 我交换使用**购物车**和**订单**这两个词，因为它们都是使用同一个文档来表示的。两者仅在文档的`state`字段上有所不同（文档状态是`CART`的表示购物车）。

```
  pricing: {
    retail: 589700,
    sale:   489700
  }
}
```

构建该文档时，很可能就是查询products集合，随后抽取出需要保存为订单条目的字段。产品中的_id、sku、slug、name和price字段应该就够了[①]。有了购物车明细文档，就可以把它upsert进订单集合了：

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state:   'CART',
            'line_items.id':
              {'$ne': ObjectId("4c4b1476238d3b4dd5003981")}
           }

update = {'$push': {'line_items': cart_item}}

db.orders.update(selector, update, true, false)
```

为了让代码更清晰一点，我分别构造了查询选择器和更新文档。更新文档将购物车明细文档塞进订单条目数组里。查询选择器中指出仅在数组中不存在特定订单条目时，更新才会成功。当然，用户第一次执行“添加到购物车”功能时，根本就没有购物车。这就是此处使用upsert的原因。upsert会根据查询选择器和更新文档里的键和值构建文档。因此，初始的upsert会产生如下订单文档：

```
{
user_id: ObjectId("4c4b1476238d3b4dd5000001"),
state:  'CART',
line_items: [{
    _id:  ObjectId("4c4b1476238d3b4dd5003981"),
    slug: "wheel-barrow-9092",
    sku:  "9092",

    name: "Extra Large Wheel Barrow",

    pricing: {
      retail: 589700,
      sale:   489700
    }
  }]
}
```

随后需要再发起一次针对性更新，确保明细数量和订单小计的正确性：

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
            state:  "CART",
            'line_items.id': ObjectId("4c4b1476238d3b4dd5003981")}

update = {$inc:
            {'line_items.$.qty': 1,
             sub_total: cart_item['pricing']['sale']
            }
         }

db.orders.update(selector, update)
```

① 在实际的电子商务应用程序中，会需要在结账时验证一下价格是否发生变化。

请注意，这里使用了$inc操作符来更新订单小计和单独条目的数量。第二条更新使用了上一节介绍的位置操作符（$），方便了不少。需要第二条更新的主要原因是要处理用户单击添加到购物车的东西已经存在于购物车中的情况。针对这种情况，第一条更新不会成功，但仍然需要调整数量和小计。因此，在两次单击手推车的“添加到购物车”功能按钮后，购物车看起来应该是这样的：

```
{
'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
'state'   : 'CART',
'line_items': [{
    _id:  ObjectId("4c4b1476238d3b4dd5003981"),
    qty:  2,
    slug: "wheel-barrow-9092",
    sku:  "9092",
    name: "Extra Large Wheel Barrow",
    pricing: {
      retail: 589700,
      sale:   489700
    }
  }],
  subtotal: 979400
}
```

现在购物车里有两部手推车了，小计上也有所体现。

还需要更多操作才能完整实现一个购物车，其中大多数都能通过一个或者多个针对性更新来实现，例如从购物车中删除一项，或者清空购物车。如果这还不明显，接下来的小节中会描述每个查询操作符，应该会让一切都清晰明了的。在实际的订单处理中，可以通过推进订单状态以及应用每个状态的处理逻辑来处理订单。下一节会演示这些内容，而且我还会解释原子文档处理和findAndModify命令。

6.3　原子文档处理

有一个工具你肯定不想错过，那就是MongoDB的findAndModify命令[①]。该命令允许对文档进行原子性更新，并在同一次调用中将其返回。因为它带来了无限可能，所以非常重要。举例来说，可以使用findAndModify来构建任务队列和状态机，随后用这些简单的构件来实现基础的事务语义，这在极大程度上扩展了能用MongoDB构建的应用程序范围。有了这些与事务类似的特性，就能在MongoDB上构造出整个电子商务站点，不仅是产品内容，还有结账和库存管理功能。

我们会通过两个实际的findAndModify命令的例子来做演示。首先展示如何处理购物车中的基本状态变迁，然后看一个更进一步的例子——管理有限的库存。

① 不同环境里，该命令的标识也会有所不同。Shell辅助方法是通过db.orders.findAndMofify这样的驼峰式大小写规则拼写来调用的，而Ruby则使用下划线：find_and_modify。更让人困惑的是核心服务器所接受的命令是findandmodify。如果需要手动发起命令，则需要使用最后一种形式。

6.3.1 订单状态变迁

所有的状态变迁都有两部分：一次查询，确保是一个合法的初始状态；一次更新，触发状态的变更。让我们跳过订单处理里的一些步骤，假设用户正要单击“现在支付”功能按钮Pay Now来授权购买。如果要在应用程序端同步授权用户的信用卡，则需要确保以下几件事：

(1) 只能授权用户在结账页面上看到的金额；

(2) 在授权过程中购物车的内容不能发生变化；

(3) 授权过程中发生错误时，要让购物车回到前一个状态；

(4) 如果信用卡授权成功，将支付信息提交到订单里，订单的状态变为SHIPMENT PENDING。

第一步是让订单进入PRE-AUTHORIZE状态。我们使用findAndModify查找用户的当前订单对象，并确保对象是CART状态的：

```
db.orders.findAndModify({
  query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
          state: "CART" },

  update: {"$set": {"state": "PRE-AUTHORIZE"},
  new: true}
})
```

如果成功，findAndModify会返回状态变迁后的订单对象。[①]一旦订单进入PRE-AUTHORIZE状态，用户就无法再编辑购物车的内容了，这是因为对购物车的所有更新总是确保CART状态。现在，处于预授权状态，我们利用返回的订单对象，重新计算各项总计。计算完毕之后，发出新的findAndModify，当新的总计和之前的一致时，将订单的状态变迁为AUTHORIZING。以下是用到的findAndModify命令：


```
db.orders.findAndModify({
  query: {user_id:  ObjectId("4c4b1476238d3b4dd5000001"),
           total: 99000,
           state: "PRE-AUTHORIZE" },

  update: {"$set": {"state": "AUTHORIZING"}}
})
```

如果第二个findAndModify失败了，那么必须将订单的状态退回为CART，并将更新后的总计信息告诉用户。但如果它成功了，那么我们就知道授权的总金额和呈现给用户的金额是一样的，也就是说可以继续进行实际的授权API调用了。应用程序现在会对用户的信用卡发起一次信用卡授权请求。如果授权失败，和之前一样，把失败记录下来，将订单退回CART状态。

但如果授权成功，把授权信息写入订单，订单流转到下一个状态，两步都在同一个findAndModify调用里完成。下面这个例子通过一个示例文档来表示接受到的授权信息，它会附加到原订单上：

① 默认情况下，findAndModify命令会返回更新前的文档。要返回修改后的文档，必须像示例中那样指定{new: true}。

```
auth_doc = {ts: new Date(),
            cc: 3432003948293040,
            id: 2923838291029384483949348,
            gateway: "Authorize.net"}
db.orders.findAndModify({
  query: {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
             state: "AUTHORIZING" },

  update: {"$set":
                 {"state": "PRE-SHIPPING"},
                  "authorization": auth_doc}
})
```

请注意，MongoDB的一些特性简化了这个事务性过程。我们可以原子性地修改任意文档，单个连接中能保证读取的一致性。最后，文档结构本身也允许这些操作来适应MongoDB提供的单文档原子性。本例中，文档结构允许将订单条目、产品、价格和用户身份都放进同一个文档里，保证只需操作一个文档就能完成销售。

本例应该让你印象深刻，也会让你感到疑惑（就像我一样），MongoDB到底能否实现多对象事务行为呢？答案是肯定的，可以通过另一个电子商务网站功能来做演示，即库存管理功能。

6.3.2　库存管理

并非所有电子商务网站都需要严格的库存管理，大多数商品都有充足的时间进货，这使得订单不用考虑当前商品的实际数量。这种情况下，管理库存就是简单地管理期望值；当库存仅有少量存货时，调整送货预期即可。

限量商品则有不同的挑战。假设正在销售指定座位的音乐会门票或者手工艺术品，这些产品是不能套期保值的，用户总是希望保证能购买到自己所选的产品。本节我将展示一种使用了MongoDB的可行解决方案。这能进一步说明findAndModify命令的创造性，以及如何明智地使用文档模型，还能演示如何实现跨多个文档的事务性语义。

建模库存的最好方法就是想象一个真实的商店。如果在一家园艺商店里，我们能看见并感受到实际库存量；很多铲子、耙子和剪刀在过道里摆成一排。要是我们拿起一把铲子放进购物车里，对其他顾客而言就少了一把铲子，其结果就是两个客户不能同时在他们的购物车里拥有同一把铲子。我们可以使用这个简单的原则来建模库存。在库存集合中为仓库里的每件实际商品保存一个对应的文档。如果仓库里有10把铲子，数据库里就有10个铲子文档。每个库存项都通过sku链接到产品上，并且拥有AVAILABLE (0)、IN_CART (1)、PRE_ORDER (2)和PURCHASED (3)这四个状态中的某个状态。

下面的代码插入了三把铲子、三把耙子和三把剪刀作为可用库存：

```
3.times do
  @inventory.insert({:sku => 'shovel',   :state => AVAILABLE})
  @inventory.insert({:sku => 'rake',     :state => AVAILABLE})
  @inventory.insert({:sku => 'clippers', :state => AVAILABLE})
end
```

我们将用一个特殊的库存获取类来管理库存。我们先看看它是如何工作的，然后深入其中，

揭示它的实现原理。

库存获取器能向购物车内添加任意产品集合。此处，我们创建了一个新订单对象与一个新的库存获取器。随后用获取器向指定订单添加了三把铲子和一把剪刀，订单由传给add_to_cart方法的订单ID指定，另外再传入两个文档指定产品和数量：

```
@order_id = @orders.insert({:username => 'kbanker', :item_ids => []})
@fetcher  = InventoryFetcher.new(:orders    => @orders,
                                 :inventory => @inventory)

@fetcher.add_to_cart(@order_id,
                     {:sku => "shovel", :qty => 3},
                     {:sku => "clippers", :qty => 1})

order = @orders.find_one({"_id" => @order_id})
puts "\nHere's the order:"
p order
```

如果某件商品添加失败，add_to_cart方法会抛出一个异常。如果执行成功，订单应该是这样的：

```
{"_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
 "username"=>"kbanker",
 "item_ids" => [BSON::ObjectId('4cdf3668238d3b6e32000001'),
               BSON::ObjectId('4cdf3668238d3b6e32000004'),
               BSON::ObjectId('4cdf3668238d3b6e32000007'),
               BSON::ObjectId('4cdf3668238d3b6e32000009')],
}
```

订单文档里会保存每件实际库存项的_id，可以像下面这样查询这些库存项：

```
puts "\nHere's each item:"
order['item_ids'].each do |item_id|
  item = @inventory.find_one({"_id" => item_id})
  p item
end
```

仔细查看每个条目，会发现它们的状态都是1，对应了IN_CART状态，而且其中还用时间戳记录了上次状态改变的时间。如果商品被放入购物车的时间太长了，稍后还可以使用这个时间戳对这些商品做过期处理。举例来说，可以规定用户有15分钟来完成将商品添加到购物车到结账的整个流程：

```
{"_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
  "sku"=>"shovel", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}

{"_id"=>BSON::ObjectId('4cdf3668238d3b6e32000009'),
  "sku"=>"clippers", "state"=>1, "ts"=>"Sun Nov 14 01:07:52 UTC 2010"}
```

如果这个InventoryFetcher的API还讲得过去，那么你应该能预感到如何实现库存管理了。findAndModify命令又在其中发挥了重要作用。本书的源代码中包含了InventoryFetcher的完整源代码及测试套件。此处我们不会仔细介绍每行代码，但会着重说明其中的三个重要方法。

首先，当传入一个要添加到购物车里的商品列表时，库存获取器会尝试将它们的状态从AVAILABLE变更为IN_CART。如果操作中有哪一步失败了（比如某项商品未能添加到购物车里），那么整个操作就会回滚。看看之前调用过的add_to_cart方法：

```
def add_to_cart(order_id, *items)
  item_selectors = []
  items.each do |item|
    item[:qty].times do
      item_selectors << {:sku => item[:sku]}
    end
  end

  transition_state(order_id, item_selectors, :from => AVAILABLE,
     :to => IN_CART)
end
```

该方法并没有完成上述功能，它只是接收要添加到购物车的具体商品并增加其数量，这样每件实际添加到购物车里的商品都能有一个库存项选择器。举例来说，以下文档表示想添加两把铲子：

```
{:sku => "shovel", :qty => 2}
```

会变成：

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

针对每件要添加到购物车里的商品，都需要一个单独的查询选择器。因此，add_to_cart方法会将库存项选择器数组传给一个名为transition_state的方法。例如，上述代码指明了状态应该从AVAILABLE变更为IN_CART：

```
def transition_state(order_id, selectors, opts={})
  items_transitioned = []

  begin
    for selector in selectors do
      query = selector.merge(:state => opts[:from])

      physical_item = @inventory.find_and_modify(:query => query,
        :update => {'$set' => {:state => opts[:to], :ts => Time.now.utc}})

      if physical_item.nil?
        raise InventoryFetchFailure
      end

      items_transitioned << physical_item['_id']

      @orders.update({:_id => order_id},
                     {"$push" => {:item_ids => physical_item['_id']}})
    end

  rescue Mongo::OperationFailure, InventoryFetchFailure
    rollback(order_id, items_transitioned, opts[:from], opts[:to])
    raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
  end

  items_transitioned.size
end
```

为了变更状态，每个选择器都有一个额外的条件{:state => AVAILABLE}，随后选择器会被传给findAndModify，如果条件匹配，则设置时间戳和库存项的新状态。transition_state方法会保存变更过状态的库存项列表，将它们的ID更新到订单里。

如果findAndModify命令执行失败并返回nil，那么会抛出一个InventoryFetchFailure异常。如果命令由于网络错误而失败，那么必然会有Mongo::OperationFailure异常，我们需要捕获该异常。这两种情况下，都要回滚之前修改过的库存项，然后抛出一个InventoryFetchFailure，其中包含无法添加的库存项SKU。随后能在应用层捕获该异常，告诉用户操作失败了。

现在就只剩下回滚的代码了：

```
def rollback(order_id, item_ids, old_state, new_state)
  @orders.update({"_id" => order_id},
                 {"$pullAll" => {:item_ids => item_ids}})

  item_ids.each do |id|
    @inventory.find_and_modify(
      :query  => {"_id" => id, :state => new_state},
      :update => {"$set" => {:state => old_state, :ts => Time.now.utc}}
    )
  end
end
```

我们使用$pullAll操作符删除了刚才添加到订单item_ids数组里的所有ID。然后遍历库存项ID列表，将每项的状态改回原来的样子。

可以将transition_state方法作为其他变更库存项状态方法的基础，要将其整合进在上一节里构建的订单流转系统应该并不困难。这就作为练习留给读者了。

你可能会问：该系统是否足够强健，能够用于生产环境？在没有了解更多详情之前，无法轻易得出结论，但可以肯定的是MongoDB提供了足够的特性，在需要类似事务的行为时，能有一个可用的解决方案。当然，没人会用MongoDB构建一个银行系统。但如果只需要某类事务行为，可以考虑使用MongoDB，尤其是想让整个应用程序运行在一个数据库上的时候。

6.4　具体细节：MongoDB 的更新与删除

要真正掌握MongoDB中的更新，需要彻底理解MongoDB的文档模型和查询语言，前几节里的例子对此很有帮助。不过，与全书的“具体细节”部分一样，我们会讨论实质性的问题。此处不仅会囊括MongoDB更新接口中每个特性的简要概述，还有多项与性能相关的说明。简单起见，后续的示例都使用JavaScript。

6.4.1　更新类型与选项

MongoDB支持针对性更新与替换更新。前者使用一个或多个更新操作符来定义，后者使用一个文档来替换匹配更新查询选择器的文档。

语法说明：更新与查询

刚接触MongoDB的用户有时会分不清更新与查询的语法。针对性更新总是由更新操作符开始的，这些操作符几乎全是动词形式的。以$addToSet操作符为例：

```
db.products.update({}, {$addToSet: {tags: 'green'}})
```

如果要为该更新增加一个查询选择器，请注意这个查询操作符在语义上起着形容词的作用，紧跟在要查询的字段名之后：

```
db.products.update({price: {$lte: 10}},
  {$addToSet: {tags: 'cheap'}})
```

基本上，更新操作符是前缀，而查询操作符通常是中缀。

请注意，如果更新文档含糊不清，更新将会失败。此处，我们将更新操作符$addToSet和替换风格的{name: "Pitchfork"}结合在一起：

```
db.products.update({}, {name: "Pitchfork", $addToSet: {tags: 'cheap'}})
```

如果目的是改变文档的名字，必须使用$set操作符：

```
db.products.update({},
  {$set: {name: "Pitchfork"}, $addToSet: {tags: 'cheap'}})
```

1. 多文档更新

默认情况下，更新操作只会更新查询选择器匹配到的第一个文档。要更新匹配到的所有文档，需要明确指定*多文档更新*（multidocument update）。在Shell里，要实现这一点，可以将update方法的第四个参数设置为true。下面展示如何为产品集合里的所有文档添加cheap标签：

```
db.products.update({}, {$addToSet: {tags: 'cheap'}}, false, true)
```

使用Ruby驱动（和大多数其他驱动）时，可以更清楚地表示多文档更新：

```
@products.update({}, {'$addToSet' => {'tags' => 'cheap'}}, :multi => true)
```

2. upsert

某项内容不存在时进行插入，存在则进行更新，这是很常见的需求。可以使用MongoDB的upsert轻松实现这一模式。如果查询选择器匹配到文档，进行普通的更新操作。如果没有匹配到文档，将会插入一个新文档。新文档的属性合并自查询选择器与针对性更新的文档。[①]

以下是在Shell中使用upsert的简单示例：

```
db.products.update({slug: 'hammer'}, {$addToSet: {tags: 'cheap'}}, true)
```

这是Ruby中等效的upsert示例：

```
@products.update({'slug' => 'hammer'},
  {'$addToSet' => {'tags' => 'cheap'}}, :upsert => true)
```

你应该已经猜到了，upsert一次只能插入或更新一个文档。在需要原子性地更新文档，以及无法确定文档是否存在时，upsert能发挥巨大的作用。6.2.3节中有一个实际的例子，描述了如何向购物车中添加产品。

① 请注意，upsert无法用于替换风格的更新操作。

6.4.2　更新操作符

MongoDB支持很多更新操作符，此处我会为每个更新操作符提供一个简单的示例。

1. 标准更新操作符

第一组是最常用的操作符，几乎能用于任意数据类型。

- $inc

可以使用$inc操作符递增或递减数值：

```
db.products.update({slug: "shovel"}, {$inc: {review_count: 1}})
db.users.update({username: "moe"}, {$inc: {password_retires: -1})
```

也可以用它加或减任意数字：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}})
```

$inc既方便又高效，因为它很少会改变文档的大小，$inc通常原地作用在数据的磁盘位置上，所以只会影响到指定的数据对。[①]

正如添加产品到购物车的示例中演示的那样，$inc能用于upsert中。例如，可以将之前的更新改为upsert：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}}, true)
```

如果不存在_id是324的文档，会用该_id创建一个新文档，文档中temp的值就是$inc的2.7435。

- $set与$unset

如果需要为文档中的特定键赋值，可以使用$set。为键赋值时，可以使用任意合法的BSON类型。也就是说以下更新都是正确的：

```
db.readings.update({_id: 324}, {$set: {temp: 97.6}})
db.readings.update({_id: 325}, {$set: {temp: {f: 212, c: 100} })
db.readings.update({_id: 326}, {$set: {temps: [97.6, 98.4, 99.1]}})
```

如果键已存在，其值会被覆盖；否则会创建一个新的键。

$unset能删除文档中特定的键。下面展示如何删除文档中的temp键：

```
db.readings.update({_id: 324}, {$unset: {temp: 1})
```

还可以在内嵌文档和数组之上使用$unset。这两种情况都要用点符号指定内部对象。如果集合中有两个文档：

```
{_id: 325, 'temp': {f: 212, c: 100}}
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

我们可以用下面的语句删除第一个文档里的华氏温标读数，以及第二个文档中的第0个元素：

```
db.readings.update({_id: 325},
  {$unset: {'temp.f': 1}})

db.readings.update({_id: 236},
  {$pop: {temps: -1})
```

6

① 当数字类型发生改变时，情况会有所不同。如果$inc造成32位整数被转换为64位整数，那么整个BSON文档会原地重写。

$set也能使用访问子文档和数组元素的点符号。

- $rename

如果要更改键名，请使用$rename：

```
db.readings.update({_id: 324}, {$rename: {'temp': 'temperature'}})
```

还可以重命名子文档：

```
db.readings.update({_id: 325}, {$rename: {'temp.f': 'temp.farenheit'}})
```

对数组使用$unset

请注意，在单个数组元素上使用$unset的结果可能与你设想的不一样。其结果只是将元素的值设置为null，而非删除整个元素。要彻底删除某个数组元素，可以用$pull和$pop操作符。

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$unset: {'temp.0': 1}})
```

2. 数组更新操作符

数组在MongoDB文档模型中的重要性是显而易见的，因此MongoDB理所当然地提供了很多专门用于数组的更新操作符。

- $push与$pushAll

如果需要为数组追加一些值，可以考虑$push和$pushAll，前者能向数组中添加一个值，而后者则支持添加一个值列表。例如，可以很方便地为铲子添加新标签：

```
db.products.update({slug: 'shovel'}, {$push: {'tags': 'tools'}})
```

如果需要在一个更新里添加多个标签，同样不成问题：

```
db.products.update({slug: 'shovel'},
  {$pushAll: {'tags': ['tools', 'dirt', 'garden']}})
```

注意，可以往数组里添加各种类型的值，不局限于标量（scalar）。上一节里，向购物车的明细条目数组里添加产品的代码就是一个很好的例子。

- $addToSet与$each

$addToSet也能为数组追加值，不过它的做法更细致：要添加的值如果不存在才执行添加操作。因此，如果铲子已经有了tools标签，那么以下更新不会修改文档：

```
db.products.update({slug: 'shovel'}, {$addToSet: {'tags': 'tools'}})
```

如果想在一个操作里向数组添加多个唯一的值，必须结合$each操作符来使用$addToSet。下面是一个示例：

```
db.products.update({slug: 'shovel'},
{$addToSet: {'tags': {$each: ['tools', 'dirt', 'steel']}}})
```

仅当$each中的值不在tags里时，才会进行添加。

● $pop

要从数组中删除元素，最简单的方法就是使用$pop操作符。如果用$push向数组中追加了一个元素，那么随后的$pop会删除最后添加的内容。虽然$pop常和$push一起出现，但也可以单独使用。如果tags数组里包含['tools', 'dirt', 'garden', 'steel']这四个值，那么下面的$pop会删除steel标签：

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': 1}})
```

$pop的语法和$unset类似，即{$pop: {'elementToRemove': 1}}，不同的是$pop还能接受-1来删除数组的第一个元素。下面展示如何从数组中删除tools标签：

```
db.products.update({slug: 'shovel'}, {$pop: {'tags': -1}})
```

可能有一个地方会让你不太满意，即无法返回$pop从数组中删除的值。尽管它的名字叫$pop，但其结果和你所熟知的栈式操作不太一样，请注意这一点。

● $pull与$pullAll

$pull的作用与$pop类似，但更高级一点。使用$pull时可以明确用值来指定要删除哪个数组元素，而不是位置。再来看看标签示例，如果要删除dirt标签，无需知道它在数组中的位置，只需告诉$pull操作符删除它就可以了：

```
db.products.update({slug: 'shovel'}, {$pull {'tags': 'dirt'}})
```

$pullAll类似于$pushAll，允许提供一个要删除值的列表。如果要删除dirt和garden标签，可以这样使用$pushAll：

```
db.products.update({slug: 'shovel'}, {$pullAll {'tags': ['dirt', 'garden']}})
```

6

3. 位置更新

在MongoDB中建模数据时通常会使用子文档数组，但在位置操作符出现之前，要操作那些子文档并非易事。位置操作符允许更新数组里的子文档，我们可以在查询选择器中用点符号指明要修改的子文档。若没有示例，理解起来比较麻烦，因此此处假设有一个订单文档，其中一部分内容是这样的：

```
{ _id: new ObjectId("6a5b1476238d3b4dd5000048"),
  line_items:  [
    { _id:  ObjectId("4c4b1476238d3b4dd5003981"),
      sku:  "9092",
      name: "Extra Large Wheel Barrow",
      quantity:  1,
      pricing:  {
        retail:  5897,
        sale:  4897,
      }
    },

    { _id:  ObjectId("4c4b1476238d3b4dd5003981"),
      sku:  "10027",
      name: "Rubberized Work Glove, Black",
      quantity:  2,
```

```
      pricing: {
        retail:  1499,
        sale:  1299,
      }
    }
  ]
}
```

假设想设置第二个明细条目的数量，把SKU为10027那条的数量设置为5。问题是你不清楚这个特定的子文档在line_items数组里的位置，甚至都不知道它是否存在。只需一个简单的查询选择器，以及一个使用了位置操作符的更新文档，这些问题就都迎刃而解了：

```
query  = {_id: ObjectId("4c4b1476238d3b4dd5003981"),
         'line_items.sku': "10027"}
update = {$set: {'line_items.$.quantity': 5}}

db.orders.update(query, update)
```

在'line_items.$.quantity'字符串里看到的$就是位置操作符。如果查询选择器匹配到了文档，那么有10027这个SKU的文档的下标就会替换位置操作符，从而更新正确的文档。

如果数据模型中包含子文档，那么你会发现在执行精细的文档更新操作时，位置操作符实在太有用了。

6.4.3　findAndModify命令

本章已经出现了很多findAndModify命令的鲜活示例，就差罗列它的选项了。在以下选项中，只有query以及update或remove是必选的。[①]

- query，文档查询选择器，默认为{}。
- update，描述更新的文档，默认为{}。
- remove，布尔值，为true时删除对象并返回，默认为false。
- new，布尔值，为true时返回修改后的文档，默认为false。
- sort，指定排序方向的文档，因为findAndModify一次只修改一个文档，sort选项能用来控制处理哪个文档。例如，可以按照{created_at: -1}来排序，处理最近创建的匹配文档。
- fields，如果只需要返回字段的子集，可以通过该选项指定。当文档很大时，这个选项很有用。能像在其他查询里一样指定字段，请查看第5章中与字段相关的示例。
- upsert，布尔值，为true时将findAndModify当做upsert对待。如果文档不存在，则创建之。请注意，如果希望返回新创建的文档，还需要指定{new: true}。

6.4.4　删除

得知删除文档的操作毫无挑战之后，你一定非常宽慰。我们可以删除整个集合，也可以向

① update与remove二选一。——译者注

remove方法传递一个查询选择器，删除集合的子集。删除全部评论是很容易的：

```
db.reviews.remove({})
```

但更常见的做法是删除特定用户的评论：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')})
```

所有对remove的调用都接受一个可选的查询选择器，用于指定要删除的文档。正如API所示，没有其他要说明的内容了。也许你会对这些操作的并发性和原子性心存疑问，下一节里我会对此做出解释。

6.4.5 并发性、原子性与隔离性

理解MongoDB中如何保证并发性是很重要的。自MongoDB v2.0起，锁策略非常粗放，靠一个全局读写锁来控制整个mongod实例。[①]这也就意味着，任何时刻，数据库只允许存在一个写线程或多个读线程（两者不能并存）。实际情况比听上去要好得多，因为这个锁策略还有一些并发优化措施。其中之一是，数据库持有一个内部映射，知道哪些文档在内存里。对于那些不在内存里的文档的读写，数据库会让步于其他操作，直到文档被载入内存。

第二个优化是写锁让步。任何写操作都可能耗时很久，所有其他的读写操作在此期间都会被阻塞。所有的插入、更新和删除都要持有写锁。插入的耗时一般不长，但更新就不一样了，比方说更新整个集合需要很久，涉及很多文档的删除操作也是如此。当前的解决方案是允许这些耗时很久的操作周期性地暂停，以便执行其他的读和写。在操作暂停时，它会自己停下来，释放锁，稍后再恢复。[②]

但在更新和删除文档时，这种暂停行为可能好坏掺半。很容易想到这种情况：希望在其他操作发生前更新或删除所有文档。在这些情况下，可以使用名为$atomic的特殊选项来避免暂停。简单地在查询选择器中添加$atomic操作符即可：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001'),
{$atomic: true}})
```

对于多文档更新，也可以做同样的处理。这迫使整个多文档更新在隔离的情况下执行完毕：

```
db.reviews.update({$atomic: true}, {$set: {rating: 0}}, false, true)
```

这个更新操作将所有评论的评分设为0。因为操作是隔离执行的，所以不会暂停，保证系统始终是一致的。[③]

6.4.6 更新性能说明

经验表明，对更新是如何作用于磁盘上的文档能有一个基本认识，有助于设计出性能更好的

① 本书翻译过程后期，MongoDB推出了2.2版本，去掉了全局的写锁。——译者注

② 当然，暂停和恢复通常发生在几毫秒内，因此我们这里不讨论极端的中断。

③ 注意，如果使用了$atomic的操作中途失败，并不会自动回滚。只有一半文档被更新，而另一半还是保持原来的值。

系统。你应该理解的第一件事是何种程度的更新能被称为“原地”更新。理想情况下，在磁盘上，更新对一个BSON文档的影响只是极小一部分，这样的性能是最好的，但事实并非总是如此。我来解释一下其中的缘由。

磁盘上的文档更新本质上分三种。第一种，也是最高效的，只发生在单值修改并且整个BSON文档的大小不改变的情况下。这通常会发生在`$inc`操作符上，因为`$inc`只会增加一个整数，该值在磁盘上的大小并不改变。如果这个整数是由`int`表示的，那么它会占用四个字节；长整数和双精度浮点数会占用八个字节。更改这些数字的值并不需要更多空间，因此磁盘上就只需重写该文档的一个值。

第二种更新会改变文档的大小和结构。BSON文档会表示为字节数组，文档的头四个字节总是存储文档的大小。因此，当在文档上使用`$push`操作符时，不仅增加整个文档的大小，还改变它的结构。这要求在磁盘上重写整个文档，这么做的效率还不算太差，但还是应该注意一下。如果在一个更新中使用了多个更新操作符，那么每个操作符都会重写一次文档。这也通常不算什么大问题，尤其是写操作发生在内存里时。但如果文档特别大，比如有4 MB左右，而你又在用`$push`向那些文档里添加值，那么服务器端就可能要做很多事情了。[①]

最后一种更新是重写文档的结果。如果文档扩大了，无法放入之前分配的磁盘空间里，那么该文档不仅要重写，而且还必须移到新的空间里。这种移动操作如果经常发生，代价还是很大的。为了降低此类开销，MongoDB会根据每个集合的情况动态调整*填充因子*（padding factor）。也就是说，如果有一个集合会发生很多要重新分配空间的更新，则会增加其内部填充因子。填充因子乘上插入文档的大小后就能得到要额外创建的空间。这能减少未来重新分配文档的数量。

要查看指定集合的填充因子，可以运行`stats`命令：

```
db.tweets.stats()
{
  "ns" : "twitter.tweets",
  "count" : 53641,
  "size" : 85794884,
  "avgObjSize" : 1599.4273783113663,
  "storageSize" : 100375552,
  "numExtents" : 12,
  "nindexes" : 3,
  "lastExtentSize" : 21368832,
  "paddingFactor" : 1.2,
  "flags" : 0,
  "totalIndexSize" : 7946240,
  "indexSizes" : {
  "_id_" : 2236416,
  "user.friends_count_1" : 1564672,
  "user.screen_name_1_user.created_at_-1" : 4145152
},
"ok" : 1 }
```

这一推文集合的填充因子是1.2，即插入100 B的文档时，MongoDB会在磁盘上分配120 B。

① 如果你打算执行很多更新操作，那么保持较小的文档是理所应当的事。

默认的填充因子是1，即不会分配额外空间。

现在，有个小小的忠告。此处讨论到的注意事项适用于数据大小超过内存总数，或者写负载极重的情况。因此如果正在为一个高流量网站构建分析系统，请适当参考本节的内容。

6.5 小结

本章讨论了很多内容。一开始要理解各种更新好像要接受的内容很多，但它们表现出的强大威力让人振奋。MongoDB的更新语言和它的查询语言一样复杂精妙。我们能像更新简单文档一样更新复杂的内嵌结构。有需要时，还可以原子性地更新文档，结合`findAndModify`构建事务性工作流。

读完本章之后，如果感觉能够自行使用其中的示例，那么你正渐渐成为一名MongoDB高手。

Part 3

第三部分

精通 MongoDB

读过本书的前两部分之后，你应该能从开发者的视角很好地理解 MongoDB 了。是时候换个角色了，在本书最后这一部分里，我们将从数据库管理员的视角来探讨 MongoDB。也就是说，这一部分将涉及与性能、部署、容错性和扩展性相关的所有内容。

要让 MongoDB 发挥出最好的性能，你必须要设计高效的查询，并且保证添加了合适的索引，而这是第 7 章将要讨论的话题。你会了解为什么索引如此重要、如何选择索引并运用在查询优化器中。另外，第 7 章还会介绍如何使用查询解释器和剖析器这些有用的工具。

第 8 章专注于复制，其中的大部分内容都在讲述副本集是如何工作的、如何明智地部署副本集以获得高可用性和自动故障转移。此外，你还会了解到如何使用复制扩展应用程序的读操作、定制写操作的耐久性。

水平扩展是现代数据库系统的“必杀技”；MongoDB 通过数据分区来实现水平扩展，这一过程称为分片。第 9 章介绍了分片理论与实践，说明何时应该使用分片、如何围绕分片设计 Schema，以及如何进行部署。

第 10 章介绍了部署与管理的细节。我们将看到与特定的硬件与操作系统相关的一些建议，并了解如何对在线 MongoDB 集群进行备份、监控和故障排查。

第 7 章

索引与查询优化

本章内容

- 基本的索引概念和理论
- 索引管理
- 查询优化

索引是非常重要的东西，有了正确的索引，MongoDB才能高效地使用硬件，为应用程序提供快速的查询。错误的索引则会导致相反的结果：慢查询、无法充分利用硬件。显而易见，想要高效使用MongoDB的人都必须理解索引。

但是，对于很多开发者而言，索引是个神秘的话题。情况不该是这样的，一旦读完本章，你应该能很好地理解索引。要介绍索引的概念，我们先从一个适当的思想实验[1]入手，然后探讨一些核心的索引概念，概述一下MongoDB索引的基础——B树数据结构。

接下来是一些实践。我们将讨论唯一性索引、稀疏索引和多键索引，为索引管理做些说明。随后，我们会深入研究查询优化，描述如何使用`explain()`和查询优化器。

7.1 索引理论

本节，我们将循序渐进地进行介绍，从一个扩展的类比开始，以概述一些MongoDB键的实现细节结尾。期间，我将定义很多重要的术语并提供示例。如果你不太了解复合键索引、虚拟内存和索引数据结构，那么阅读本节将会受益匪浅。

7.1.1 思想实验

要理解索引，你需要在脑中有个画面，这里建议想象一本食谱，不是普通的食谱，而是一本5000页的厚重食谱，其中包含针对各种场合、菜肴和季节的精美食谱，在家就能找到所有的配料。这是本终极食谱，让我们称它*The Cookbook Omega*。

虽然这可能是所有食谱中最好的一本，但却有两个小问题。第一，所有的食谱是乱序的，第

① 思想实验即在现实中未做到的，使用想象力进行的实验。——译者注

3475页是Australian Braised Duck，而第2页则是Zacatecan Tacos。

这还不是很要紧，关键问题是这本食谱没有索引！

下面是你要问自己的第一个问题：没有索引，如何在*The Cookbook Omega*中找到Rosemary Potatoes？唯一的选择是一页页翻过去，直到找到为止。如果它在第3973页，你得要翻多少页啊！最坏的情况下，假设它在最后一页，你需要把整本书都翻一遍！

这真令人抓狂，解决方案就是构建一个索引。

你可以想到多种食谱查找方法，其中食谱的名字可能是个不错的起点。如果建立一个按字母排列的食谱名称列表，随后是其所在页码，那么就按食谱名称对本书建立索引了。其中的条目可能是下面这样的：

- Tibetan Yak Soufflé: 45
- Toasted Sesame Dumplings: 4,011
- Turkey à la King: 943

只要知道食谱名字（哪怕只是名字的开头几个字母），就能通过该索引快速找到书中的任意食谱。如果你只希望按照这种方式来检索食谱，那就已经完事了。

但这是不现实的。比方说，你还会希望根据储藏室里的食材查找食谱，或者是根据菜肴来进行查找。针对这些情况，你需要更多的索引。

这就产生了第二个问题。只有一个基于食谱名称的索引，如何才能找到所有的鸡肉（chicken）相关的食谱呢？缺少合适的索引，你仍然需要翻阅整本食谱——5000页。在根据食材或者菜肴进行检索时都是如此。

为此，你需要构建另一个索引，这次是对食材进行索引。在这个索引里，按字母顺序排列食材，每个食材都指向所有包含它的食谱所在的页码。最基本的食材索引是这样的：

- Cashews: 3, 20, 42, 88, 103, 1,215...
- Cauliflower: 2, 47, 88, 89, 90, 275...
- Chicken: 7, 9, 80, 81, 82, 83, 84...
- Currants: 1,001, 1,050, 2,000, 2,133...

这是你所希望的索引吗？是不是很有用？

如果只是需要指定食材的食谱清单，这个索引就够用了。但如果还希望在查找时包含任意其他与食谱相关的信息，还是要进行“扫描”——一旦知道菜花（cauliflower）的页码，你要翻到每一页找到食谱的名字以及菜肴类型。这比翻遍整本书要好，但还远远不够。

例如，几个月前，你无意间在*The Cookbook Omega*里发现了一个很棒的鸡肉料理食谱，但却忘了它的名字。目前为止，有两个索引，一个是食谱名称的索引，另一个是食材的索引。是否能将两者结合起来，找到被你遗忘的鸡肉食谱呢？

实际上，这是不可能的。如果从食谱名称的索引入手，但却不记得名字，检索这个索引只比翻阅全书好一点。从食材入手，则需要检查一系列页码，但这些页码无法插入基于食谱名称的索引。因此，这种情况下只能使用一个索引，本例中食材的索引更有用一些。

每个查询一个索引

用户通常认为一个查询里要查找两个字段，可以针对它们分离索引。有一个现成的算法：查找每个索引里匹配项的页码，针对同时匹配两个索引的食谱扫描它们页码的并集。会有不少匹配不上的页码，但还是能减少扫描的总数。一些数据库实现了这个算法，但MongoDB没有。就算它实现了，使用复合索引来查找两个字段总是会比我刚才描述的算法效率更高。请记住，每个查询中数据库只会使用一个索引，如果要对多个字段进行查询，请确保有这些字段的复合索引。

那该怎么办？幸好，我们有应对之道，答案在于使用复合索引。

到目前为止你所建立的是单键索引：它们都只对食谱的一个键进行索引。现在要为*The Cookbook Omega*构建一个新的索引，不同之处是这次要使用两个键。类似的使用多个键的索引称为复合索引（compound index）。

该复合索引依次使用了食材与食谱名称。可以这样来标记它：`ingredient-name`。其中的部分内容如图7-1所示。

Cashews
　Cashew Marinade
　　1,215
　Chicken with Cashews
　　88
　Rosemary-Roasted Cashews
　　103

Cauliflower
　Bacon Cauliflower Salad
　　875
　Lemon-baked Cauliflower
　　89
　Spicy Cauliflower Cheese Soup
　　47

Currants
　Creamed Scones with Currants
　　2,000
　Fettuccini with Glazed Duck
　　2,133
　Saffron Rice with Currants
　　1,050

图7-1　食谱中的复合索引

这个索引的值对人而言是显而易见的。现在你可以根据食材进行查找，大致定位要找的食谱，哪怕只记得名称的开头部分。对机器而言，它同样很有价值，不用扫描拥有该食材的全部食谱名称了。如果有几百个（或者几千个）鸡肉料理食谱，就像*The Cookbook Omega*一样，该复合索引

尤其有用。你知道是为什么吗?

请注意一点:复合索引中的顺序是很有讲究的。假设将上述索引翻转为name-ingredient,它能替代我们之前用的复合索引吗?

明显不能!使用新索引,只要知道名称,搜索就一定会定位到一个食谱,书中的一页。如果是要查找含有香蕉(banana)食材的Cashew Marinade食谱,那么它就能确定不存在这个食谱。但现实情况恰恰相反:你知道食材,却不知道名称。

*The Cookbook Omega*现在有三个索引:recipe(食谱)、ingredient(食材)和ingredient-name(食材–食谱名称)。也就是说可以安全地去掉ingredient这个单键索引了。为什么?因为对某一食材的检索可以使用ingredient-name。如果你知道食材,可以遍历该复合索引,获得包含它的食谱的页码列表。再仔细看看该索引的示例,想想其中的原因。

本节的目的是为那些需要对索引有更进一步认识的读者提供一个隐喻。从中,你能认识到一些简单的经验法则,如下。

(1) 索引能显著减少获取文档所需的工作量。没有合适的索引,实现查询的唯一途径就是线性扫描整个文档,直到满足查询条件为止。这通常就是扫描整个集合。

(2) 解析查询时只会使用一个单键索引。[①]对于包含多个键(比如食材和食谱名称)的查询,包含这些键的复合索引能更好地解析查询。

(3) 如果有ingredient-cuisine(食材–菜肴)索引,可以去掉ingredient索引,也应该这么做。更抽象一点,如果有一个a-b的复合索引,那么仅针对a的索引就是冗余的。[②]

(4) 复合索引里键的顺序是很重要的。

请记住,这个比喻只能用到这一步。它是用来理解索引的一个模型,并不等于MongoDB中索引的工作方式。下一节里,我们会详细说明刚列出的几点内容,仔细探究MongoDB里的索引。

7.1.2 核心索引概念

上一节讲到了很多核心索引概念,本节和本章剩下的部分会对它们做详细说明。

1. 单键索引

单键索引中的每一项都对应了被索引文档里的一个值。默认的_id索引就是一个很好的例子,由于这个字段上有索引,可以根据它快速地获取文档。

2. 复合键索引

到目前为止,MongoDB中每个查询就使用一个索引。[③]但是你经常需要对多个属性进行查询,希望这些查询能尽可能高效一点。例如,假设你在本书电子商务示例的products集合上构建了

① 使用$or操作符的查询是个例外。但通常情况下,只能使用一个索引,MongoDB也更倾向于此。

② 也有例外情况。如果b是一个多键索引,同时拥有a-b和a索引还是有意义的。

③ 偶尔也有例外。举例来说,带有$or的查询里,每个$or查询的子句都能使用不同的索引,但每个子句本身只能使用一个索引。

两个索引：一个基于manufacturer（制造商）字段，另一个基于price（价格）字段。如此一来，你创建了两个截然不同的数据结构，在遍历时的顺序如图7-2所示。

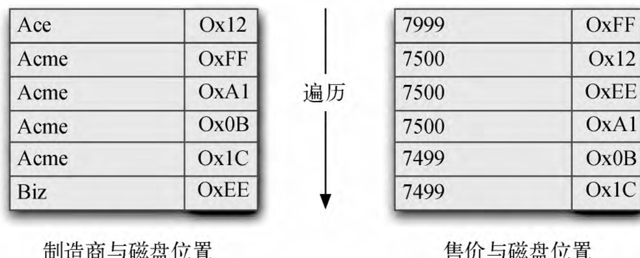


图7-2　单键索引遍历

现在，假设查询是这样的：

```
db.products.find({'details.manufacturer': 'Acme',
                  'pricing.sale': {$lt: 7500}})
```

这条查询的意思是找到所有Acme生产的售价低于75.00美元的产品。如果使用manufacturer或者price字段上的单键索引发起查询，那么只能用上其中的一个索引。查询优化器会选择两者中更高效的那个，但都无法给出理想的结果。要用这些索引满足查询，必须分别遍历这两个数据结构，抓取匹配数据的磁盘位置，再计算它们的并集。MongoDB目前并不支持这种做法，因为此时使用复合索引效率更高。

复合索引就是每一项都由多个键组合而成的索引。如果要构建一个基于manufacturer和price的复合键索引，排序后的表示如图7-3所示。



图7-3　复合键索引遍历

要实现你的查询，查询优化器只需找到索引里第一条制造商是Acme、价格是75.00美元的索引项。随后简单地扫描连续的索引项，当制造商不再是Acme时停止。这样就能取到查询结果了。

结合使用索引和查询时，有两件事需要注意。第一，在索引里键的顺序很重要。如果声明了一个复合索引，价格是第一个键，制造商位于其后，那么查询效率很差。很难想通吗？看看图7-4里此类索引的结构吧。

7999 - Acme	OxFF
7500 - Ace	OxEE
7500 - Acme	Ox12
7500 - Biz	OxA1
7499 - Acme	Ox0B
7499 - Acme	Ox1C

遍历

价格与制造商，以及磁盘位置

图7-4　键顺序相反的复合索引

必须按照键的出现顺序进行比较。很遗憾，这个索引没办法轻松地跳至所有Acme产品，因此实现之前那条查询的唯一办法是查看每件价格小于75.00美元的产品，然后只取出Acme制造的产品。为了便于理解，假设你的集合里有100万个产品，价格都低于100.00美元，按价格均匀分布。在这种情况下，执行该查询要扫描750 000个索引项。相比之下，使用原来的复合索引，即制造商在价格之前，扫描的索引项数量就等于要返回的索引项数量。这是因为一旦找到Acme - 7500这一项，剩下的就是简单的顺序扫描了。

因此，在复合索引里键的顺序很重要。清楚了这一点之后，第二件要明白的事情是为什么选择这样的顺序。从图里看还是很明显的，但还可以换个方式来看这个问题。再仔细看下查询：有两个查询项指定了不同的匹配类型。在制造商字段上，希望精确匹配一项；但在价格字段上，希望匹配一个值范围，从7500开始。一般来说，一个查询里有一项要精确匹配，另一项指定了一个范围，在使用复合索引时，范围匹配的那个键放在第二个位置上。在查询优化的章节里我们还会看到这条规则。

3. 索引效率

索引对良好的查询性能来说是必不可少的，但每个新索引都会带来一些小的维护成本。其原因是显而易见的，每当向集合添加文档时，都必须修改集合上的所有索引，以加入新的文档。因此，如果一个集合上有10个索引，每次插入时就都要做10次独立的结构修改。对于所有写操作都是如此，无论是删除文档还是更新指定文档的索引键。

对于读密集型应用而言，索引的成本一般都是合理的，你只要认识到索引还是会引入一定开销，必须谨慎选择即可。这意味着确保所有索引都被用到，没有一个索引是多余的。可以在剖析应用程序的查询时部分落实这项工作，我会在本章的后续内容里描述这个过程。

此处还有另一个问题需要考虑：就算拥有正确的索引，还是有可能得不到快速的查询，索引和数据集无法全部放入内存时就会发生这种情况。

回想第1章，MongoDB使用`mmap()`系统调用告诉操作系统将所有数据文件映射到内存里。基于这点，操作系统会按照名为页（page）的4 KB块[①]将数据文件换入换出内存，包含所有文档、集合与索引。在请求指定页的数据时，操作系统必须保证该页在内存里。如果不在，会抛出页错误（page fault）异常，告诉内存管理器从磁盘上把页加载到内存里。

① 4 KB的页大小是标准值，但并非普遍适用。

有了充足的内存，所有使用中的数据文件最终都会被加载到内存里。当那块内存发生改变时，比如执行写操作时，那些改变会被操作系统异步地刷到磁盘上，而写操作很快，是直接发生在内存里的。数据完全装入内存是最为理想的状态，因为磁盘访问的数量会降到最低程度。但如果使用中的数据集无法全部装入内存，就该出现页错误了。也就是说操作系统会频繁访问磁盘，大大减缓读写操作。最坏的情况下，数据大小远远大于可用内存，这时任何读写操作的数据都必须到磁盘上做页交换。这种情况称为**颠簸**（thrashing），会导致性能严重下滑。

还好这种情况相对容易避免。最起码要保证索引都能放入内存；对于为何避免创建无用索引如此重要，这就是原因之一。拥有额外的索引，就会要求更多的内存来维护那些索引。同样道理，每个索引应该只包含它需要的键：有时会用到三键复合索引，但请注意它要比简单的单键索引占用更多的空间。

理想情况下，索引和使用中的数据集都能放入内存。但评估部署时需要有多少内存并非易事。你可以通过查看`stats`命令的结果来了解总的索引大小。但要找到**工作集**（working set）大小却没这么容易，因为每个应用程序都不一样。工作集通常是查询与更新的全部数据的子集。举例来说，假设你有100万用户，只有一半是活跃用户，那么工作集就是用户集合的一半大小。如果全部都是活跃用户，那么工作集就等于整个数据集。

在第10章，我们会重温工作集的概念，了解诊断硬件相关性能问题的具体手段。就目前而言，只需知道添加新索引有潜在的成本，关注索引与工作集大小与内存的比率，这能帮你在数据增长时维护良好的性能。

7.1.3　B树

前文提到过，MongoDB内部使用B树来表示索引。B树无处不在（见http://mng.bz/wQfG），至少从20世纪70年代后期开始就流行于数据库记录和索引中。[①]如果你使用过其他数据库系统，那么可能已经熟知使用B树的各种情况了。这很好，你可以将之前的大多数索引相关的知识有效地利用起来。如果不太了解B树，也没关系，本节将介绍与使用MongoDB最为相关的概念。

B树有两个显著特点，并因此成为了数据库索引的理想选择。第一，它们能用于多种查询，包括精确匹配、范围条件、排序、前缀匹配和仅用索引的查询。第二，在添加和删除键的时候，它们仍能保持平衡。

我们会看到一棵简单的B树，讨论一些应该牢记在心的原则。想象有一个用户的集合，在姓氏（last name）和年龄（age）字段上创建了一个复合索引。[②]结果B树的抽象表述可能是图7-5这样的。

你一定已经猜到了，B树是一个树型数据结构。树中的每一个节点都能包含多个键。在示例中，根节点包含两个键，每个都以BSON对象的形式来表示`users`集合中被索引的值。在读取了根节点的内容之后，你能看到两个文档的键，分别表示姓氏Edwards和Perry，年龄是21岁和18岁。

---

① MongoDB中B树仅用于索引；集合存储为**双向列表**（doubly-linked list）。

② 在姓氏与年龄上创建索引有点牵强，但却能很好地阐明一些概念。

每个键都包含了两个指针：一个指向它所属的数据文件，另一个指向子节点。此外，节点自己还指向另一个节点，其值小于本节点的最小值。

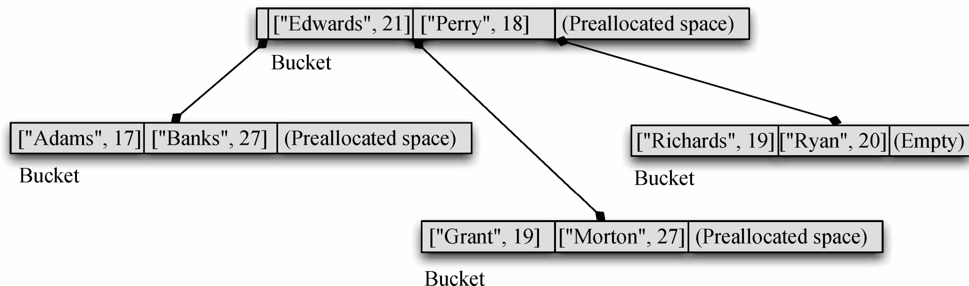


图7-5　B树结构示例

有件事情需要注意，每个节点都有一些留空的空间（不是用来伸缩的）。在MongoDB的B树实现里，新节点会被分配8192字节，也就是说实际上每个节点都能包含数百个键。这取决于索引键的平均大小；本例中，平均键大小可能在30字节左右。MongoDB v2.0里键最大可以是1024字节。每个键有额外的18字节，每个节点再多40字节，其结果就是每个节点能容纳170个键[①]。

这很有关系，因为用户经常想知道为什么索引是这个大小。你现在知道了每个节点是8 KB，可以估算出每个节点能容纳多少键。计算时，请牢记：在默认情况下，B树节点的内容通常有意维持在60%左右。

如果理解了上述内容，除了这个粗略的B树心智模型，你还应该记住一些东西，例如索引是如何使用空间及如何进行维护的：再提醒一下，索引是有代价的。请谨慎选择索引。

7.2　索引实践

了解了这么多理论之后，现在来细化一下MongoDB中索引的概念。然后，我们将深入讨论索引管理的一些细节。

7.2.1　索引类型

MongoDB中的所有索引底层都使用相同的数据结构，但可以有很多不同的属性。尤其是唯一性索引、稀疏索引和多键索引，它们都很常用，本节会详细介绍它们。[②]

1. 唯一性索引

要创建唯一性索引，设置unique选项即可：

```
db.users.ensureIndex({username: 1}, {unique: true})
```

① (8192 − 40) / (30 + 18) = 169.8。

② 注意，MongoDB还支持空间索引，但因为它的用途太专业了，我会在附录E中单独进行说明。

唯一性索引保证了集合中所有索引项的唯一性。如果要向本书示例应用程序的用户集合users插入一个文档，其中的用户名已经被索引过了，那么插入会失败，抛出如下异常：

```
E11000 duplicate key error index:
  gardening.users.$username_1  dup key: { : "kbanker" }
```

如果使用驱动，那么只有在使用驱动的安全模式执行插入时才会捕获该异常。第3章中有对此的相关讨论。

如果集合上需要唯一性索引，通常在插入数据前先创建索引会比较好。提前创建索引，能在一开始就保证唯一性约束。在已经包含数据的集合上创建唯一性索引时，会有失败的风险，因为集合里可能已经存在重复的键了。存在重复键时，创建索引会失败。

如果真有需要在一个已经建好的集合上创建唯一性索引，你有几个选择。首先是不停地重复创建唯一性索引，根据失败消息手动删除包含重复键的文档。如果数据不重要，还可以通过dropDups选项告诉数据库自动删除包含重复键的文档。举个例子，如果用户集合users里已经有数据了，而且你并不介意删除包含重复键的文档，可以像下面这样发起索引创建命令：

```
db.users.ensureIndex({username: 1}, {unique: true, dropDups: true})
```

请注意，要保留哪个重复键的文档是不确定的，因此在使用时要特别小心。

2. 稀疏索引

索引默认都是密集型的。也就是说，在一个有索引的集合里，每个文档都会有对应的索引项，哪怕文档中没有被索引键也是如此。例如，回想一下电子商务数据模型里的产品集合，假设你在产品属性category_ids上构建了一个索引。现在假设有些产品没有分配给任何分类，对于每个无分类的产品，category_ids索引中仍会存在像这样的一个null项。可以这样查询null值：

```
db.products.find({category_ids: null})
```

在查询缺少分类的所有产品时，查询优化器仍然能使用category_ids上的索引定位对应产品。

但是有两种情况使用密集型索引会不太方便。一种是希望在并非出现在集合所有文档内的字段上增加唯一性索引时。举例来说，你明确希望在每个产品的sku字段上增加唯一性索引。但是出于某些原因，假设产品在还未分配sku时就加入系统了。如果sku字段上有唯一性索引，而你希望插入多个没有sku的产品，那么第一次插入会成功，但后续插入都会失败，因为索引里已经存在一个sku为null的项了。这种情况下密集型索引并不适合，你所需要的是*稀疏索引*（sparse index）。

在稀疏索引里，只会出现被索引键有值的文档。如果想创建稀疏索引，指定{sparse: true}就可以了。例如，可以像下面这样在sku上创建一个唯一性稀疏索引：

```
db.products.ensureIndex({sku: 1}, {unique: true, sparse: true})
```

另一种适用稀疏索引的情况：集合中大量文档都不包含被索引键。例如，假设允许对电子商务网站进行匿名评论。这种情况下，半数评论都可能缺少user_id字段，如果那个字段上有索引，那么该索引中一半的项都会是null。出于两个原因，这种情况的效率会很差。第一，这会增加

索引的大小。第二，在添加和删除带null值user_id字段的文档时也要求更新索引。

如果很少（或不会）对匿名评论进行查询，那么可以选择在user_id上构建一个稀疏索引。设置sparse选项同样非常简单：

```
db.reviews.ensureIndex({user_id: 1}, {sparse: true})
```

现在就只有那些通过user_id字段关联了用户的评论才会被索引。

3. 多键索引

在之前的几章里，你已经见过好多索引字段的值是数组的例子了。[①]正是名为多键索引（multikey index）的东西让这些成为可能，它允许索引中的多个条目指向相同文档。我们可以举个简单的例子说明一下，假设有一个产品文档，包含几个标签：

```
{ name: "Wheelbarrow",
  tags: ["tools", "gardening", "soil"]
}
```

如果在tags上创建索引，标签数组里的每个值都会出现在索引里。也就是说，对数组中任意值的查询都能用索引来定位文档。多键索引背后的理念是这样的：多个索引项或键最终指向同一个文档。

MongoDB中的多键索引总是处于激活状态。被索引字段只要包含数组，每个数组值都会在索引里有自己的位置。

合理使用多键索引是正确设计MongoDB Schema时必不可少的一环，这在第4章到第6章的例子里已经很明显了；附录B的设计模式部分还会提供更多的示例。

7.2.2 索引管理

要在MongoDB中管理索引，你现有的知识可能还稍有不足。本节我们将详细介绍索引的创建和删除，并讨论与压紧（compaction）和备份相关的问题。

1. 索引的创建与删除

到目前为止，你已经创建了很多索引，因此对索引的创建语法应该并不陌生。在Shell或者所选语言里简单地调用索引创建辅助方法，会在特殊的system.indexes集合中添加一个文档定义新的索引。

虽然通常情况下使用辅助方法创建索引会更方便一些，但也可以手工插入一个索引说明（辅助方法就是这么做的）。你只需确保指定了以下这些最起码的键：ns、key与name。ns是命名空间，key是要索引的字段或字段的组合，name是用来指向索引的名字。此处还能指定一些额外选项，比方说sparse。例如，让我们在users集合上创建一个索引：

```
spec = {ns: "green.users", key: {'addresses.zip': 1}, name: 'zip'}
db.system.indexes.insert(spec, true)
```

如果插入操作没有返回错误，那么索引就创建完毕了，可以查询system.indexes集合进行确认：

① 举例来说，分类ID。

7

```
db.system.indexes.find()
{ "_id" : ObjectId("4d2205c4051f853d46447e95"), "ns" : "green.users",
  "key" : { "addresses.zip" : 1 }, "name" : "zip", "v" : 1 }
```

如果你在使用MongoDB v2.0或后续版本，会看到一个额外的键`v`。这个版本字段能用于未来内部索引格式的变更，但应用程序开发者不用太在意它。

要删除索引，你可能会觉得就是删除`system.indexes`里的索引文档，但这个操作是被禁止的，你必须使用数据库命令`deleteIndexes`删除索引。和创建索引一样，删除索引也有辅助方法可用，如果希望直接运行该方法，也没有问题。该命令接受一个文档作为参数，其中包含集合名称、要删除的索引名称或者用*来删除所有索引。要手工删除刚刚创建的索引，使用如下命令：

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

大多数情况下，只需简单地使用Shell里的辅助方法创建和删除索引：

```
use green
db.users.ensureIndex({zip: 1})
```

然后可通过`getIndexSpecs()`方法来检查索引说明：

```
> db.users.getIndexSpecs()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
        "ns" : "green.users",
        "name" : "_id_"
    },
    {
        "v" : 1,
        "key" : {
            "zip" : 1
        },
        "ns" : "green.users",
        "name" : "zip_1"
    }
]
```

最后，可以使用`dropIndex()`方法删除索引。请注意，必须提供上述定义里的索引名称：

```
use green
db.users.dropIndex("zip_1")
```

以上是基本的索引创建与删除，想知道索引创建以后该做些什么，请往下读。

2. 索引的构建

大多数时候，你会在把应用程序正式投入使用之前添加索引，这允许随着数据的插入增量地构建索引。但在两种情况下，你可能会选择相反的过程。第一种情况是在切换到生产环境之前需要导入大量数据。举例来说，你想将应用程序迁移到MongoDB，需要从数据仓库导入用户信息。你可以事先在用户数据上创建索引，但在数据导入之后再创建索引能从一开始就保证理想的平衡性和密集的索引，也能将构建索引的净时间降到最低。

第二种（更显而易见的）情况发生在为新查询进行优化的时候。

无论为什么要创建新索引，这个过程都很难让人愉快起来。对于大数据集，构建索引可能要花好几个小时，甚至好几天。但你可以从MongoDB的日志里监控索引的构建过程。来看一个例子。先声明要构建的索引：

```
db.values.ensureIndex({open: 1, close: 1})
```

声明索引时要小心

由于这个步骤太容易了，所以也很容易在无意间触发索引构建。如果数据集很大，构建会花很长时间。在生产环境里，这简直就是梦魇，因为没办法中止索引构建。如果发生了这种情况，你将不得不故障转移到从节点上——如果有从节点的话。最明智的建议是将索引构建当做某类数据库迁移来看待，确保应用程序的代码不会自动声明索引。

索引的构建分为两步。第一步，对要索引的值排序。经过排序的数据集在插入到B树时会更高效。注意，排序的进度会以已排序文档数和总文档数的比率来进行显示：

```
[conn1] building new index on { open: 1.0, close: 1.0 } for stocks.values
    1000000/4308303  23%
    2000000/4308303  46%
    3000000/4308303  69%
    4000000/4308303  92%
    Tue Jan  4 09:59:13 [conn1]    external sort used : 5 files  in 55 secs
```

第二步，排序后的值被插入索引中。进度显示方式与第一步相同，完成之后，完成索引构建所用的时间会显示出来，作为插入`system.indexes`的耗时：

```
1200300/4308303  27%
    2227900/4308303  51%
    2837100/4308303  65%
    3278100/4308303  76%
    3783300/4308303  87%
    4075500/4308303  94%
Tue Jan  4 10:00:16 [conn1] done building bottom layer, going to commit
Tue Jan  4 10:00:16 [conn1] done for 4308303 records 118.942secs
Tue Jan  4 10:00:16 [conn1] insert stocks.system.indexes 118942ms
```

除了查看MongoDB的日志，还可以通过Shell的`currentOp()`方法检查构建索引的进度：[①]

```
> db.currentOp()
{
  "inprog" : [
    {
      "opid" : 58,
      "active" : true,
      "lockType" : "write",
      "waitingForLock" : false,
```

① 注意，如果是在MongoDB Shell里开始索引构建的，则必须打开一个新的Shell并发地运行`currentOp`。关于`db.currentOp()`的更多内容，详见第10章。

```
        "secs_running" : 55,
        "op" : "insert",
        "ns" : "stocks.system.indexes",
        "query" : {

        },
        "client" : "127.0.0.1:53421",
        "desc" : "conn",
        "msg" : "index: (1/3) external sort 3999999/4308303 92%"
      }
    ]
}
```

最后一个字段msg描述了构建进度。还要注意lockType，它说明索引构建用了写锁，也就是说其他客户端此时无法读写数据库。如果发生在生产环境里，这无疑是很糟糕的，这也是长时间索引构建让人抓狂的原因。我们接下来会看到两个可行的解决方案。

- 后台索引

如果是在生产环境里，经不住这样暂停数据库访问的情况，可以指定在后台构建索引。虽然索引构建仍会占用写锁，但构建任务会停下来允许其他读写操作访问数据库。如果应用程序大量使用MongoDB，后台索引会降低性能，但在某些情况下这是可接受的。例如，假设你知道可以在应用程序流量最低的时间窗口内完成索引的构建，那么这时后台索引会是个不错的选择。

要在后台构建索引，声明索引时需要指定{background: true}。可以像下面这样在后台构建之前的索引：

```
db.values.ensureIndex({open: 1, close: 1}, {background: true})
```

- 离线索引

如果生产数据集太大，无法在几小时内完成索引，这时就需要其他方案了。通常这会涉及让一个副本节点下线，在该节点上构建索引，随后让其上的数据与主节点同步。一旦完成数据同步，将该节点提升为主节点，再让另一个从节点下线，构建它自己的索引。该策略假设你的复制oplog够大，能避免离线节点的数据在索引构建过程中变得过旧。下一章会详细讨论复制，应该能帮你计划这样的迁移过程。

3. 备份

因为索引很难构建，所以你可能会希望为它们做备份，可惜并非所有备份方法都包含索引。举例来说，你可能想使用mongodump和mongorestore，但这些工具仅保存了集合和索引声明。也就是说，当运行mongorestore时，所备份的所有集合上声明的索引都会被重新创建一遍。如果数据集很大，那么构建索引所消耗的时间也是无法接受的。

因此，如果想要在备份中包含索引，需要直接备份MongoDB的数据文件。第10章里有更具体的讨论，以及常用的备份操作指南。

4. 压紧

如果应用程序会大量更新现有数据，或者执行很多大规模删除，其结果就是索引的碎片化程度很高。虽说B树会自己合并，但这并非总能抵消大量删除的影响。碎片过多的索引大小远超你对指定数据集大小的预期，也会让索引使用更多内存。这些情况下，你可能希望重建一个或多个

索引：可以删除并重新创建单个索引，或者运行reIndex命令（它会重建指定集合上的所有索引）：

```
db.values.reIndex();
```

在重建索引时要小心：在重建过程中该命令会占用写锁，让你的MongoDB实例暂时无法使用。重建最好是在线下完成，就像之前提到的在从节点上构建索引一样。请注意第10章里将要介绍的compact命令，它也会重建集合上的索引。

7.3 查询优化

查询优化是识别慢查询、找出它们为什么慢、逐步让它们变快的过程。本节里，我们会依次看到查询优化过程中的每一步，当你读完本节之后，基本就能找出MongoDB里所有的问题查询了。

在深入之前，我必须提醒一下，本节出现的技术不能解决所有查询的性能问题。慢查询的原因千奇百怪，糟糕的应用程序设计、不恰当的数据模型、硬件配置不够都是常见的原因，处理这些问题要耗费大量时间。此处我们会看到通过重新组织查询以及构建有效的索引来进行优化的方法。我还将介绍其他途径，以便你在上述手段不奏效时尝试一下。

7.3.1 识别慢查询

如果感觉基于MongoDB的应用程序变慢了，那么就该着手剖析查询语句了。任何严谨的应用程序设计方法中都应该包含对查询语句的审核；考虑到MongoDB中这一切都是如此简单，没有理由不这么做。虽然每个应用程序对查询语句的要求各有不同，但可以保守地进行假设：对于大多数应用而言，查询都不该超过100 ms。这个假设被固化在了MongoDB的日志里，无论什么操作（包括查询在内），只要超过100 ms就会输出一条警告。因此，要识别慢查询，第一时间就该看日志。

到目前为止，我们的数据集都很小，无法生成执行时间超过100 ms的查询。所以随后的例子里，我们将使用一组由NASDAQ日汇总数据组成的数据集。如果你也希望能执行这些查询，需要将它们放到本地数据库里。要导入它，首先从http://mng.bz/ii49下载压缩包，然后将其解压到一个临时文件夹里。你将看到如下输出：

```
$ unzip stocks.zip
Archive:  stocks.zip
   creating: dump/stocks/
  inflating: dump/stocks/system.indexes.bson
  inflating: dump/stocks/values.bson
```

最后，用以下命令将数据还原到数据库里：

```
$ mongorestore -d stocks -c values dump/stocks
```

股票数据集很大，而且方便使用。针对某个NASDAQ上市股票的子集，有从1983年开始25年的数据，每天一个文档，记录每日的最高价、最低价、收盘价和成交量。有了如此数量的集合文档，很容易就能生成一条日志警告。试着查询第一条谷歌股价：

```
db.values.find({"stock_symbol": "GOOG"}).sort({date: -1}).limit(1)
```

你会注意到这条查询执行了一段时间。如果查看MongoDB的日志，会看到预期中的慢查询警告。下面就是一段示例输出：

```
Thu Nov 16 09:40:26 [conn1] query stocks.values
          ntoreturn:1 scanAndOrder reslen:210 nscanned:4308303
          { query: { stock_symbol: "GOOG" }, orderby: { date: -1.0 } }
          nreturned:1 4011ms
```

其中包含大量信息，在讨论`explain()`时，我们会研究其中所有内容的含义。现在，如果仔细阅读这段消息，应该能抽取出最重要的部分：这是针对`stocks.values`的查询；执行的查询选择器包含匹配`stock_symbol`以及排序；最关键的可能是这个查询花了4 s（4011 ms）之久。

一定要设法处理这样的警告。它们太关键了，值得你时不时地筛查MongoDB的日志。可以通过`grep`轻松实现筛查：

```
grep -E '([0-9])+ms' mongod.log
```

如果100 ms的阈值太高了，可以通过`--slowms`服务器选项降低这个值。要是把慢查询定义为执行时间超过50 ms，那么用`--slowms 50`来启动`mongod`。

当然，筛查日志还不够彻底。你可以通过日志检查慢查询，但这个过程太粗糙了，应该将其作为预发布或生产环境中的一种“健康检查”。要在那些慢查询成为问题之前识别它们，你需要一个更精确的工具，MongoDB内置的查询剖析器正是你所需要的。

使用剖析器

要识别慢查询，离不开MongoDB内置的剖析器。剖析功能默认是关闭的，让我们先把它打开。在MongoDB Shell中，输入以下命令：

```
use stocks
db.setProfilingLevel(2)
```

先选择要剖析的数据库，因为剖析总是针对某个特定数据库的。随后将剖析级别设置为2，这是最详细的级别；它告诉剖析器将每次的读和写都记录到日志里。还有一些其他选项。若只要记录慢（100 ms）操作，可以将剖析级别设置为1。要彻底禁用剖析器，将级别设置为0。如果只想在日志里记录耗时超过一定毫秒阈值的操作，可以像下面这样将毫秒数作为第二个参数：

```
use stocks
db.setProfilingLevel(1, 50)
```

一旦开启了剖析器，就可以执行查询了。让我们再运行一条对股票数据库的查询，找出数据集中最高的收盘价：

```
db.values.find({}).sort({close: -1}).limit(1)
```

剖析结果会保存在一个特殊的名为`system.profile`的固定集合里。你是否还记得，固定集合的大小是确定的，数据会像环一样写入其中，一旦集合达到最大尺寸，新文档会覆盖最早的文档。`system.profile`被分配了128 KB，因此确保剖析数据不会消耗太多资源。

你可以像查询任何固定集合那样查询`system.profile`。举例来说，查询所有耗时超过150 ms的语句：

```
db.system.profile.find({millis: {$gt: 150}})
```

因为固定集合保持了自然插入顺序，可以用`$natural`操作符进行排序，以便先显示最近的结果：

```
db.system.profile.find().sort({$natural: -1}).limit(5)
```

回到刚才的查询语句，结果集里应该会有大致这样一条内容：

```
{ "ts" : ISODate("2011-09-22T22:42:38.332Z"),
"op" : "query", "ns" : "stocks.values",
"query" : { "query" : { }, "orderby" : { "close" : -1 } },
"ntoreturn" : 1, "nscanned" : 4308303, "scanAndOrder" : true,
"nreturned" : 1, "responseLength" : 194, "millis" : 14576,
"client" : "127.0.0.1", "user" : "" }
```

又是一条慢查询：耗时将近15 s！除了执行时间，其中还包含所有在MongoDB慢查询警告中出现查询的信息，足够进行更深一步的排查了，而下一节里就会讲到这个话题。

但在继续之前，还得再说一些与剖析策略有关的内容。先使用较粗的设置，然后不断细化，用这种方式来使用剖析器就挺不错的。首先保证没有查询超过100 ms，然后将阈值降低到75 ms，以此类推。开启剖析器之后，你会想把应用程序测一遍，最起码把每个读写操作都执行一遍。如果考虑的周到一些，就必须在真实条件下执行那些操作，数据大小、查询负载和硬件都应该能代表应用程序的生产环境。

查询剖析器十分有用，但要将它发挥到极致，你还得有条理。比起生产环境，最好能在开发过程中找到慢查询，不然补救的成本会大很多。

7.3.2　分析慢查询

有了MongoDB的剖析器，可以很方便地找到慢查询。要知道这些查询为什么慢会更麻烦一点，因为这个过程中可能还要求有点“侦察工作”。正如前文所述，慢查询的原因是多种多样的。走运的话，加个索引就能解决慢查询。在更复杂的情况里，可能不得不重新安排索引、重建数据模型，或者升级硬件。但是，总是应该先看看最简单的情况，本节就与此相关。

最简单的情况里，问题的根本原因是缺少索引、索引不当或者查询不理想。可以在慢查询上运行`explain`来确认原因。现在，让我们来了解下具体的做法。

1. 使用并了解**EXPLAIN()**

MongoDB的`explain`命令提供了关于指定查询路径的详细信息。[①]让我们仔细地看一看，对上一节里运行的最后一条查询执行`explain`能收集到什么信息。要在Shell中运行`explain`，只需在查询后附上`explain()`方法调用：

```
db.values.find({}).sort({close: -1}).limit(1).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 4308303,
  "nscannedObjects" : 4308303,
```

① 可以回忆一下我在第2章中介绍的`explain`，当时只是简单地介绍。本节我将提供完整的命令说明及其输出。

7

```
  "n" : 1,
  "scanAndOrder" : true,
  "millis" : 14576,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : { }
}
```

millis字段指出该查询耗时超过14 s，其原因很明显。请看nscanned的值，它表明查询引擎必须扫描4 308 303个文档才能完成查询。现在，在values集合上运行count：

```
db.values.count()
4308303
```

扫描的文档数与集合中的文档总数一致，也就是说执行了一次全集合扫描。如果你希望查询返回集合里的全部文档，这倒不是一件坏事。但是如果仅需返回一个文档，正如explain中的n所示，那这就成问题了。一般来说，希望n的值与nscanned的值尽可能接近。在进行集合扫描时，情况往往不是这样的。cursor字段指明你在使用BasicCursor，这只能说明在扫描集合本身而非索引。

scanAndOrder字段进一步解释了查询缓慢的原因，当查询优化器无法使用索引来返回排序结果集时，它就会出现。因此，本例中不仅查询引擎需要扫描集合，还要求手动对结果集进行排序。

如此之差的性能是无法接受的，好在应对之道比较简单。你只需要在close字段上构建一个索引。现在就动手，然后重新发起查询：①

```
db.values.ensureIndex({close: 1})
db.values.find({}).sort({close: -1}).limit(1).explain()
{
  "cursor" : "BtreeCursor close_1 reverse",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : {
    "close" : [
      [
        {
          "$maxElement" : 1
        },
        {
          "$minElement" : 1
        }
      ]
    ]
  }
}
```

① 注意，索引的构建可能需要几分钟。

差距太大了！这次的查询处理只用了不到1 ms。通过cursor字段可以看到，正在使用名为close_1的索引上的BtreeCursor，而且在倒序迭代索引。在indexBounds字段里，可以看到特殊值$maxElement和$minElement，它们说明查询横跨了整个索引。此时查询优化器经过B树的最右边才找到最大键，然后再沿路返回。因为限制了返回集为1，在找到了最大元素后查询就完成了。当然，由于索引是有序保存索引项的，就没有必要再进行scanAndOrder所指定的手工排序了。

如果在查询选择器中使用了经过索引的键，就会看到输出中有些许不同之处。来看看查询收盘价大于500的查询语句的explain输出：

```
> db.values.find({close: {$gt: 500}}).explain()
{
  "cursor" : "BtreeCursor close_1",
  "nscanned" : 309,
  "nscannedObjects" : 309,
  "n" : 309,
  "millis" : 5,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "indexBounds" : {
    "close" : [
      [
        500,
        1.7976931348623157e+308
      ]
    ]
  }
}
```

扫描的文档数仍然与返回的文档数相同（n与nscanned是一致的），这是理想状态。请注意，在索引边界的指定方式上，此处与前者有所不同。这里没有使用$maxElement与$minElement键，边界是实际值。下限是500，上限实际是无限大。这些值必须和正在查询的值使用相同的数据类型；在查询的是数字，所以这里的索引边界是数字。如果要查询一系列字符串，那么边界就是字符串。[①]

7

在继续之前，请自己在查询上运行explain()，要注意n和nscanned之间的不同。

2. MongoDB的查询优化器与hint()

查询优化器是MongoDB中的一部分，如果存在可用的索引，它会为给定查询选择一个最高效的索引。在为查询选择理想的索引时，查询优化器使用了一套相当简单的规则：

(1) 避免scanAndOrder。如果查询中包含排序，尝试使用索引进行排序；

(2) 通过有效的索引约束来满足所有字段——尝试对查询选择器里的字段使用索引；

(3) 如果查询包含范围查找或者排序，那么对于选择的索引，其中最后用到的键需能满足该范围查找或排序。

① 如果觉得这无法理解，请回忆一下，某个指定索引能包含多种数据类型的键。因此，查询结果总是会被限制在查询所使用的数据类型中。

如果某个索引能满足以上所有这些条件，那么它就会被视为最佳索引并予以使用。要是有多个最佳索引，则任意选择其一。可以遵循这条经验：如果能为查询构建最优索引，查询优化器的工作能更轻松些。为此，请尽力而为。

让我们来看一个查询，它完全满足索引（和查询优化器）。回顾股票数据集，假设要执行如下查询，获取所有大于200的谷歌收盘价：

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}})
```

该查询的最优索引同时包含这两个键，但其中把close键放在最后以便执行范围查询：

```
db.values.ensureIndex({stock_symbol: 1, close: 1})
```

如果执行查询，会看到这两个键都被用到了，索引边界也和预想的一样：

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain()
{
  "cursor" : "BtreeCursor stock_symbol_1_close_1",
  "nscanned" : 730,
  "nscannedObjects" : 730,
  "n" : 730,
  "millis" : 1,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "stock_symbol" : [
      [
        "GOOG",
        "GOOG"
      ]
    ],
    "close" : [
      [
        200,
        1.7976931348623157e+308
      ]
    ]
  }
}
>
```

这是本条查询的最优explain输出：n与nscanned的值相同。现在再来考虑一下没有索引能完美运用于查询之上的情况。例如，没有{stock_symbol: 1, close: 1}索引，但是在那两个字段上分别建有索引。通过getIndexKeys()列出索引，会看到：

```
db.values.getIndexKeys()
[ { "_id" : 1 }, { "close" : 1 }, { "stock_symbol" : 1 } ]
```

因为查询中同时包含stock_symbol和close两个键，没有很明显的索引可用。这时就该查询优化器出马了，它所用的试探方式比想象的要简单得多，完全基于nscanned的值。换言之，优化器会选择扫描索引项最少的索引。查询首次运行时，优化器会为每个可能有效适用于该查询

的索引创建查询计划，随后并行运行各个计划[①]，nscanned值最低的计划胜出。优化器会停止那些长时间运行的计划，将胜出的计划保存在来，以便后续使用。

你可以发起查询并运行explain()来查看实际的过程。首先，删除复合索引{stock_symbol: 1, close: 1}，在这些键上构建单独的索引：

```
db.values.dropIndex("stock_symbol_1_close_1")
db.values.ensureIndex({stock_symbol: 1})
db.values.ensureIndex({close: 1})
```

将true作为参数传递给explain()方法，这能将查询优化器尝试的计划列表包含在输出里。输出见代码清单7-1。

代码清单7-1 用explain(true)查看查询计划

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain(true)
{
  "cursor" : "BtreeCursor stock_symbol_1",
  "nscanned" : 894,
  "nscannedObjects" : 894,
  "n" : 730,
  "millis" : 8,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
    "stock_symbol" : [
      [
        "GOOG",
        "GOOG"
      ]
    ]
  },
  "allPlans" : [
    {
      "cursor" : "BtreeCursor close_1",
      "indexBounds" : {
        "close" : [
          [
            100,
            1.7976931348623157e+308
          ]
        ]
      }
    },
    {
      "cursor" : "BtreeCursor stock_symbol_1",
      "indexBounds" : {
        "stock_symbol" : [
          [
            "GOOG",
```

① 严格地说，这些计划是交错在一起的。

```
            "GOOG"
          ]
        ]
      }
    },
    {
      "cursor" : "BasicCursor",
      "indexBounds" : {
      }
    }
  ]
}
```

你马上能发现查询计划选择了{stock_symbol: 1}索引来实现查询。输出的下方，allPlans键指向一个列表，其中还包含了两个额外的查询计划：一个使用{close: 1}索引，另一个用BasicCursor扫描集合。

优化器拒绝集合扫描的原因显而易见，但不选择{close :1}索引的原因却不明显。可以通过hint()找到答案，hint()能强迫查询优化器使用某个特定索引：

```
query = {stock_symbol: "GOOG", close: {$gt: 100}}
db.values.find(query).hint({close: 1}).explain()
{
  "cursor" : "BtreeCursor close_1",
  "nscanned" : 5299,
  "n" : 730,
  "millis" : 36,
  "indexBounds" : {
    "close" : [
      [
        200,
        1.7976931348623157e+308
      ]
    ]
  }
}
```

nscanned的值是5299，这比之前扫描的894项要多得多，完成查询的时间也证实了这一点。

剩下的就是要理解查询优化器是如何缓存它所选择的查询计划，并让其过期的。毕竟，你不会希望优化器对每条查询都并行运行所有计划。

在发现了一个成功的计划之后，会记录下查询模式（query pattern）、nscanned的值以及索引说明。针对刚才的查询，所记录的结构是这样的：

```
{ pattern: {stock_symbol: 'equality', close: 'bound',
  index: {stock_symbol: 1},
  nscanned: 894 }
```

查询模式记录下了每个键的匹配类型，你正请求对stock_symbol的精确匹配（相等），对close的范围匹配（边界）[①]。只要新的查询匹配此模式，就会使用该索引。

① 也许你会对此感兴趣，共有三种范围匹配类型：上界（upper）、下界（lower）以及上下界（upper-and-lower）。查询模式还包含各种排序。

但这一信息不应该是永久的，实际情况也是如此。在发生以下事件之后优化器会自动让计划过期。

- 对集合执行了100次写操作。
- 在集合上增加或删除了索引。
- 虽然使用了缓存的查询计划，但工作量大于预期。此处，“工作量大”的标准是nscanned超过缓存的nscanned值的10倍。

发生最后一种事件时，优化器会立即开始交错执行其他查询计划，也许另一个索引会更高效。

7.3.3　查询模式

此处列举了几种常见的查询模式，以及它们所使用的索引。

1. 单键索引

要讨论单键索引，请回忆一下为股票集合的收盘价创建的索引{close: 1}，该索引能用于以下场景。

- 精确匹配

举例来说，要精确匹配所有收盘价是100的条目：

```
db.values.find({close: 100})
```

- 排序

可以对被索引字段排序。例如：

```
db.values.find({}).sort({close: 1})
```

本例中的排序没有查询选择器，除非真的打算迭代整个集合，否则你可能会希望再增加一个限制。

- 范围查询

针对某个字段进行范围查询，在同一字段上带不带排序都可以。例如，查询所有大于或等于100的收盘价：

```
db.values.find({close: {$gte: 100})
```

如果对同一个键增加排序子句，优化器仍能使用相同的索引：

```
db.values.find({close: {$gte: 100}).sort({close: 1})
```

2. 复合键索引

复合键索引稍微复杂一点，但它们的用法与单键索引类似。有一点要牢记，针对每个查询，复合键索引只能高效适用于单个范围或排序。仍然是股价的例子，想象一个三复合键索引{close: 1，open: 1，date: 1}，可能会有以下几种场景。

- 精确匹配

精确匹配第一个键、第一和第二个键，或者第一、第二和第三个键，按照这个顺序：

```
db.values.find({close: 1})
db.values.find({close: 1, open: 1})
db.values.find({close: 1, open: 1, date: "1985-01-08"})
```

7

- 范围匹配

精确匹配任意一组最左键（包含空），随后对其右边紧邻的键进行范围查询或者排序。于是，以下所有的查询对于该三键索引而言都是十分理想的：

```
db.values.find({}).sort({close: 1})
db.values.find({close: {$gt: 1}})

db.values.find({close: 100}).sort({open: 1})
db.values.find({close: 100, open: {$gt: 1}})

db.values.find({close: 1, open: 1.01, date: {$gt: "2005-01-01"}})
db.values.find({close: 1, open: 1.01}).sort({date: 1})
```

3. 覆盖索引

如果你从未听说过*覆盖索引*（covering index，也称*索引覆盖*），那么从一开始就要意识到这个术语并不恰当。覆盖索引不是一种索引，而是对索引的一种*特殊用法*。如果查询所需的所有数据都在索引自身之中，那就可以说索引能覆盖该查询。覆盖索引查询也称*仅使用索引的查询*（index-only query），因为不用引用被索引文档本身就能实现这些查询，这能带来性能的提升。

MongoDB中能很方便地使用覆盖索引，简单地选择存在于单个索引里的字段集合，排除掉`_id`字段（因为这个字段几乎不会出现在正使用的索引中）。下面这个例子里用到了上一节创建的三复合键索引：

```
db.values.find({open: 1}, {open: 1, close: 1, date: 1, _id: 0})
```

如果对它执行`explain()`，你会看到其中标识为`indexOnly`的字段被设为了`true`。这说明查询结果是由索引而非实际集合数据提供的。

查询优化总是针对特定应用程序的，但是我希望本节的理念和技术能帮助你更好地调整查询。通过观察和实验进行调整总是行之有效的方法。要养成习惯剖析并解释你的查询，在此过程中，你会了解查询优化器鲜为人知的一面，并能保证应用程序的查询性能。

7.4　小结

本章的内容可能有点多，因为索引实在是个非常大的主题。如果对本章中阐述的一些内容还不是很清楚，没什么关系。至少你了解了一些技术，可以用来检查索引并避免慢查询，掌握了足够多的知识以进一步学习。鉴于索引和查询优化的复杂性，从现在起，你最好的老师可能就是那些简单的实验。

第8章 复　　制

本章内容

- ❑ 基本复制概念
- ❑ 管理副本集并处理故障转移
- ❑ 副本集连接、写关注、读扩展与标签

复制（replication）是大多数数据库管理系统的重要功能，因为故障是不可避免的。如果希望生产数据在故障之后也保持可用状态，务必要确保生产数据库被部署在多台服务器上。在发生故障时，复制能提供高可用性与灾难恢复能力。

本章开始时，我会介绍复制的概念并讨论其主要使用场景，通过深入研究副本集来探讨MongoDB的复制功能。最后，我将描述如何使用驱动连接到复制后的MongoDB集群、如何使用写关注（write concern）、如何在副本间实现读操作的负载均衡。

8.1　复制概述

复制就是在多台服务器上分布并管理数据库服务器。MongoDB提供了两种复制风格：主从复制和副本集。两种方式都是在一个主节点进行写操作（写入的数据被异步地同步到所有的从节点上），并从节点上读取数据。

主从复制和副本集使用了相同的复制机制，但是副本集还能保证自动故障转移：如果主节点由于某些原因下线了，可能的话，会自动将一个从节点提升为主节点。副本集还提供了其他增强，比如更易于恢复和更高级的部署拓扑。出于这些原因，现在已经没什么有说服力的理由使用简单的主从复制了。[①]副本集也因此是生产部署环境的推荐复制策略；因此，本章的大部分内容都是副本集的说明和例子，对主从复制只做了一个简单的概述。

8.1.1　为什么复制很重要

所有数据库都对其运行环境中的故障很敏感，而复制提供了一种抵御故障的机制。这里所指

① 只有一种情况需要选择MongoDB的主从复制，即需要超过11个从节点时，因为副本集不能包含12个以上的成员。

的故障都有哪些？下面是一些常见场景。

- 应用程序与数据库之间的网络连接丢失。
- 计划停机，但服务器没有按照预定计划重新上线。任何机构的服务器都一定会安排偶尔停一下机，而其停机结果不太好预测。一次简单的重启至少能让数据库服务器下线几分钟，但问题是重启完成之后又会发生什么呢？新安装的软件或硬件经常会让操作系统无法正常启动。
- 断电。虽然大多数现代化数据中心都提供了冗余电源，但无法避免数据中心内部的用户错误、大范围局部暂时限制用电或者停电造成数据库服务器关闭。
- 数据库服务器硬盘故障。硬盘通常都只有几年的平均无故障时间，它比你想象的更容易发生故障。[①]

除了抵御外部故障，复制对于MongoDB的耐久性来说也很重要。如果运行时没有开启Journaling日志，遇到非正常关闭，无法保证MongoDB的数据文件不受破坏。没有Journaling日志，就必须时刻运行复制，这才能确保在一个节点意外关闭时能有一份数据文件的正确副本。

当然，就算*开启*了Journaling日志也应该使用复制，毕竟你追求高可用性和快速故障转移。此时Journaling日志会迅速完成恢复，因为只需简单地回放Journaling日志就能让故障节点重新上线。相比从现有副本*重新同步*（resyncing）或手动复制副本的数据文件，这要快得多。

无论是否开启Journaling日志，MongoDB的复制功能都会极大地增强整个数据库的可靠性，因此强烈推荐使用该功能。

8.1.2　复制的使用场景

你可能会感到很惊讶，复制数据库的用途居然能有这么多。尤其是复制能帮助进行冗余、故障转移、维护以及负载均衡。下面，让我们简单地看一些使用场景。

复制主要是用来做冗余的。本质上要保证复制节点与主节点保持同步。这些副本可以和主节点位于同一数据中心里，也可以分布在不同地理位置用于容灾。因为复制是异步的，任何节点间的网络延迟或分区（partition）都不会影响主节点的性能。作为另一种形式的冗余，复制节点也可以与主节点保持一定的延时，万一用户无意间删除了一个集合，或者应用程序不知怎么的破坏了数据库，这还能提供一些防御措施。一般情况下，这些操作都会被立即复制；延时副本让管理员有时间做出反应，也许还能挽救他们的数据。

有一点很重要：虽然它们是一种冗余，但副本不是备份的替代品。备份是数据库在过去某个特定时间的快照，但副本总是最新的。在一些情况下，数据集过大让备份显得不切实际，但通常来说，备份是种明智的做法，就算用了复制也推荐进行备份。

复制的另一个使用场景是故障转移。你希望系统高可用，但仅在拥有冗余节点，并且在紧急情况下能切换到这些节点时，才能实现高可用。MongoDB的副本集通常都能方便地自动实现这

---

① 可以在谷歌的“Failure Trends in a Large Disk Drive Population”（http://research.google.com/archive/disk_failures.pdf）中看到硬盘故障率的详细分析。

种切换。

除了提供冗余与故障转移，复制还可以简化维护工作，它允许你在主节点以外的节点上执行开销很大的操作。例如，通常都会在从节点上进行备份，不给主节点带来额外的负载，避免停机。另一个例子是构建大索引，因为构建索引的开销很大，可以先在某个从节点上构建索引，然后主从切换，再在新的从节点上构建索引。

最后，复制能让你在副本间均衡读负载。对于那些读负载占绝对比重的应用程序而言，这是扩展MongoDB最简单的途径。话虽如此，但如果出现以下场景，请不要用从节点来扩展读操作。

- 所分配的硬件无法处理给定的负载。以我上一章里提到的工作集为例，如果使用的工作数据集远大于可用内存，那么向从节点发送随机读请求仍然可能造成大量磁盘访问，导致慢查询。
- 读写比超过50%。诚然，这个比例有点主观，但将它作为起始值还是挺合适的。此处的问题是主节点上的所有写操作最终也会写入从节点，把读操作导向正在处理大量写入的从节点有时会减缓复制过程，并不会提高读吞吐量。
- 应用程序要求一致性读。从节点的复制是异步进行的，因此无法保证一定能读到主节点上最新写入的数据。在某些极端情况下，从节点可能延迟几个小时。

因此，你能通过复制来均衡读负载，但仅限于特定场景。如果需要进行扩展，又出现了以上某种情况，那么你需要不同的策略，包括分片、升级硬件或两者兼而有之。

8.2　副本集

副本集是对主从复制的一种完善，也是推荐的MongoDB复制策略。我们会从配置一个示例副本集开始，然后描述复制是如何工作的，这些知识对于诊断线上问题是极为重要的。最后会讨论一些高级配置细节、故障转移与恢复，还有最佳部署实践。

8.2.1　配置

最小的推荐副本集配置由三个节点组成。其中两个节点是一等的、持久化mongod实例，两者都能作为副本集的主节点，都有完整的数据副本。集合里的第三个节点是*仲裁节点*，不复制数据，只是中立观察者。正如其名所示，仲裁节点是进行仲裁的：在要求故障转移时，仲裁节点会帮助选出新的主节点。图8-1描绘了要配置的副本集。

先为副本集里的每个成员创建数据目录：

```
mkdir /data/node1
mkdir /data/node2
mkdir /data/arbiter
```

接下来，分别为每个成员启动独立的mongod。因为要在同一台机器上运行这些进程，最好在独立的终端窗口里启动各个mongod：

```
mongod --replSet myapp --dbpath /data/node1 --port 40000
mongod --replSet myapp --dbpath /data/node2 --port 40001
mongod --replSet myapp --dbpath /data/arbiter --port 40002
```

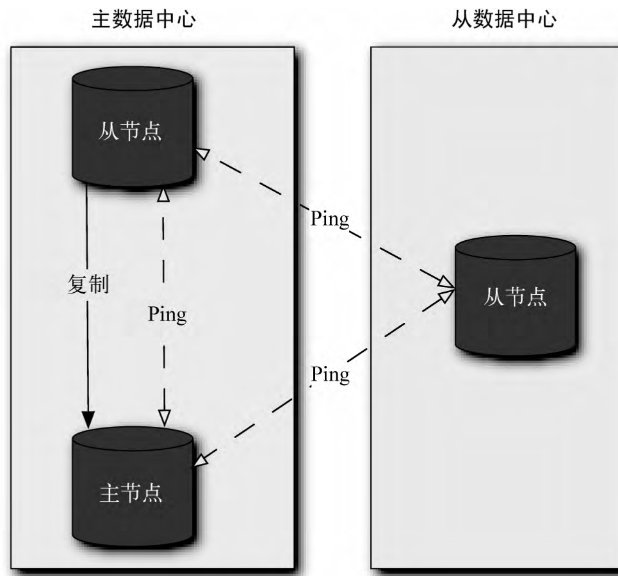


图8-1 由一个主节点、一个从节点和一个仲裁节点组成的基本副本集

如果查看mongod的日志输出，注意到的第一件事是错误消息（提示找不到配置）。这完全正常：

```
[startReplSets] replSet can't get local.system.replset
  config from self or any seed (EMPTYCONFIG)
[startReplSets] replSet info you may need to run replSetInitiate
```

继续下一步，需要配置副本集。先连接到一个刚启动的非仲裁节点的mongod上。这里的例子都是在本地运行mongod进程的，因此将通过本地主机名来进行连接，本例中是arete。

连接后运行rs.initiate()命令：

```
> rs.initiate()
{
    "info2" : "no configuration explicitly specified -- making one",
    "me" : "arete:40000",
    "info" : "Config now saved locally.  Should come online in about a minute
     .",
    "ok" : 1
}
```

一分钟左右，你就能拥有一个单成员的副本集了。现在再通过rs.add()添加其他两个成员：

```
> rs.add("localhost:40001")
{ "ok" : 1 }
> rs.add("arete.local:40002", {arbiterOnly: true})
{ "ok" : 1 }
```

注意，在添加第二个节点时指定了arbiterOnly参数，以此创建一个仲裁节点。不久之后（1分钟内），所有的成员就都在线了。要获得副本集状态的摘要信息，可以运行db.isMaster()命令：

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "arete:40001",
    "arete:40000"
  ],
  "arbiters" : [
    "arete:40002"
  ],
  "primary" : "arete:40000",
  "maxBsonObjectSize" : 16777216,
  "ok" : 1
}
```

`rs.status()`方法能提供更详细的系统信息，可以看到每个节点的状态信息。下面是完整的状态信息：

```
> rs.status()
{
    "set" : "myall",
    "date" : ISODate("2011-09-27T22:09:04Z"),
    "myState" : 1,
    "members" : [
        {
            "_id" : 0,
            "name" : "arete:40000",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "optime" : {
                "t" : 1317161329000,
                "i" : 1
            },
            "optimeDate" : ISODate("2011-09-27T22:08:49Z"),
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "arete:40001",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 59,
            "optime" : {
                "t" : 1317161329000,
                "i" : 1
            },
            "optimeDate" : ISODate("2011-09-27T22:08:49Z"),
            "lastHeartbeat" : ISODate("2011-09-27T22:09:03Z"),
            "pingMs" : 0
        },
```

```
        {
            "_id" : 2,
            "name" : "arete:40002",
            "health" : 1,
            "state" : 7,
            "stateStr" : "ARBITER",
            "uptime" : 5,
            "optime" : {
                "t" : 0,
                "i" : 0
            },
            "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
            "lastHeartbeat" : ISODate("2011-09-27T22:09:03Z"),
            "pingMs" : 0
        }
    ],
    "ok" : 1
}
```

除非你的MongoDB数据库里包含很多数据，否则副本集应该能在30 s内上线。在此期间，每个节点的`stateStr`字段应该会从`RECOVERING`变为`PRIMARY`、`SECONDARY`或`ARBITER`。

就算副本集的状态"宣称"复制已经在运行了，你可能还是希望能看到一些证据。因此，接下来在Shell里连接到主节点，插入一个文档：

```
$ mongo arete:40000
> use bookstore
switched to db bookstore
> db.books.insert({title: "Oliver Twist"})
> show dbs
admin (empty)
bookstore 0.203125GB
local 0.203125GB
```

初始的复制几乎是立即发生的。在另一个终端窗口中开启一个新的Shell实例，这次要指向从节点。查询刚才插入的文档，应该有如下输出：

```
$ mongo arete:40001
> show dbs
admin (empty)
bookstore 0.203125GB
local 0.203125GB
> use bookstore
switched to db bookstore
> db.books.find()
{ "_id" : ObjectId("4d42ebf28e3c0c32c06bdf20"), "title" : "Oliver Twist" }
```

如果复制确实如显示的那样已经在运作了，那就说明已经成功配置了副本集。

能实实在在地看到复制让人觉得很满意，但也许自动故障转移会更有趣一些。现在就来做点测试。要模拟网络分区需要点技巧，所以我们会选择一个简单的方法，杀掉一个节点。你可以杀掉从节点，这只会停止复制，剩余的节点仍旧保持其当前状态。如果希望看到系统状态发生改变，就需要杀掉主节点。标准的CTRL-C或`kill -2`就能办到这点。你还可以连上主节点，在Shell里运行`db.shutdownServer()`。

一旦杀掉了主节点，从节点会发现检测不到主节点的“心跳”了，随后会把自己“选举”为主节点。这样的“选举”是可行的，因为原始节点中的大多数节点（仲裁者节点和原始的从节点）仍能ping到对方。以下是从节点日志的片段：

```
[ReplSetHealthPollTask] replSet info arete:40000 is down (or slow to respond)
Mon Jan 31 22:56:22 [rs Manager] replSet info electSelf 1
Mon Jan 31 22:56:22 [rs Manager] replSet PRIMARY
```

如果连接到新的主节点上检查副本集状态，你会发现无法访问到老的主节点：

```
> rs.status()
{
      "_id" : 0,
      "name" : "arete:40000",
      "health" : 1,
      "state" : 6,
      "stateStr" : "(not reachable/healthy)",
      "uptime" : 0,
      "optime" : {
        "t" : 1296510078000,
        "i" : 1
      },
      "optimeDate" : ISODate("2011-01-31T21:43:18Z"),
      "lastHeartbeat" : ISODate("2011-02-01T03:29:30Z"),
      "errmsg": "socket exception"
}
```

故障转移后，副本集就只有两个节点了。因为仲裁节点没有数据，只要应用程序只和主节点通信，它就能继续运作。[1]即使如此，复制停止了，现在不能再做故障转移了。老的主节点必须恢复。假设它是正常关闭的，可以让它再度上线，它会自动以从节点的身份重新加入副本集。你可以试一下，现在就重启老的主节点。

以上就是副本集的完整概述，毫无悬念，你会觉得其中一些细节有点棘手。在接下来的两节里，你会看到副本集实际是如何运作的，了解它的部署、高级配置以及如何处理生产环境中可能出现的复杂场景。

8.2.2 复制的工作原理

副本集依赖于两个基础机制：oplog和“心跳”（heartbeat）。oplog让数据的复制成为可能，而“心跳”则监控健康情况并触发故障转移，后续将看到这些机制是如何轮流运作的。你应该已经逐渐开始理解并能预测副本集的行为了，尤其是在故障的情况下。

1. 关于oplog

oplog是MongoDB复制的关键。oplog是一个固定集合，位于每个复制节点的`local`数据库里，记录了所有对数据的变更。每次客户端向主节点写入数据，就会自动向主节点的oplog里添加一个条目，其中包含了足够的信息来再现数据。一旦写操作被复制到某个从节点上，从节点的oplog

① 应用程序有时会查询从节点来做读扩展。如果是这样，此类故障就会导致读故障。因此在设计应用程序时要时刻把故障转移放在心头。本章末尾会有更多相关内容。

也会保存一条关于写入的记录。每个oplog条目都由一个BSON时间戳进行标识，所有从节点都使用这个时间戳来追踪它们最后应用的条目。[①]

为了更好地了解其原理，让我们仔细看看真实的oplog以及其中记录的操作。先在Shell里连接到上一节启动的主节点，切换到local数据库：

```
> use local
switched to db local
```

local数据库里保存了所有的副本集元数据和oplog。当然，这个数据库本身不能被复制。正如其名，local数据库里的数据对本地节点而言是唯一的，因此不该复制。

如果查看local数据库，你会看到一个名为oplog.rs的集合，每个副本集都会把oplog保存在这个集合里。你还会看到一些系统集合，以下就是完整的输出：

```
> show collections
me
oplog.rs
replset.minvalid
slaves
system.indexes
system.replset
```

replset.minvalid包含了指定副本集成员的初始同步信息，system.replset保存了副本集配置文档。me和slaves用来实现写关注（本章最后会介绍）。system.indexes是标准索引说明容器。

我们先把精力集中在oplog上，查询与上一节里你所添加图书文档相关的oplog条目。为此，输入如下查询，结果文档里会有四个字段，我们将依次讨论这些字段：

```
> db.oplog.rs.findOne({op: "i"})
{ "ts" : { "t" : 1296864947000, "i" : 1 }, "op" : "i", "ns" :
  "bookstores.books", "o" : { "_id" : ObjectId("4d4c96b1ec5855af3675d7a1"),
  "title" : "Oliver Twist" }
}
```

第一个字段是ts，保存了该条目的BSON时间戳。这里特别要注意，Shell是用子文档来显示时间戳的，包含两个字段，t是从纪元开始的秒数，i是计数器。也许你会觉得可以像下面这样来查询这个条目：

```
db.oplog.rs.findOne({ts: {t: 1296864947000, i: 1}})
```

实际上，这条查询返回null。要在查询中使用时间戳，需要显式构造一个时间戳对象。所有的驱动都有自己的BSON时间戳构造器，JavaScript也是如此。可以这样做：

```
db.oplog.rs.findOne({ts: new Timestamp(1296864947000, 1)})
```

回到那条oplog条目上，第二个字段op表示操作码（opcode），它告诉从节点该条目表示了什么操作，本例中的i表示插入。op后的ns标明了有关的命名空间（数据库和集合），o对插入操作而言包含了所插入文档的副本。

---

① BSON时间戳是一个唯一标识符，由从纪元算起的秒数和一个递增的计数器值构成。

在查看oplog条目时，你可能会注意到，对于那些影响多个文档的操作，oplog会将各个部分都分析到位。对于多项更新和大批量删除来说，会为每个影响到的文档创建单独的oplog条目。例如，假设你向集合里添加了几本狄更斯的书：

```
> use bookstore
db.books.insert({title: "A Tale of Two Cities"})
db.books.insert({title: "Great Expectations"})
```

现在集合里有四本书，让我们通过一次多项更新来设置作者的名称：

```
db.books.update({}, {$set: {author: "Dickens"}}, false, true)
```

在oplog里会出现什么呢？

```
> use local
> db.oplog.$main.find({op: "u"})
{ "ts" : { "t" : 1296944149000, "i" : 1 }, "op" : "u",
"ns" : "bookstore.books",
"o2" : { "_id" : ObjectId("4d4dcb89ec5855af365d4283") },
"o" : { "$set" : { "author" : "Dickens" } } }

{ "ts" : { "t" : 1296944149000, "i" : 2 }, "op" : "u",
"ns" : "bookstore.books",
"o2" : { "_id" : ObjectId("4d4dcb8eec5855af365d4284") },
"o" : { "$set" : { "author" : "Dickens" } } }

{ "ts" : { "t" : 1296944149000, "i" : 3 }, "op" : "u",
"ns" : "bookstore.books",
"o2" : { "_id" : ObjectId("4d4dcbb6ec5855af365d4285") },
"o" : { "$set" : { "author" : "Dickens" } } }
```

如你所见，每个被更新的文档都有自己的oplog条目。这种正规化是更通用策略中的一部分，它会保证从节点总是能和主节点拥有一样的数据。要确保这一点，每次应用的操作都必须是幂等的——一个指定的oplog条目被应用多少次都无所谓；结果总是一样的。其他多文档操作的行为是一样的，比如删除。你可以试试不同的操作，看看它们在oplog里最终是什么样的。

要取得oplog当前状态的基本信息，可以运行Shell的`db.getReplicationInfo()`方法：

```
> db.getReplicationInfo()
{
    "logSizeMB" : 50074.10546875,
    "usedMB" : 302.123,
    "timeDiff" : 294,
    "timeDiffHours" : 0.08,
    "tFirst" : "Thu Jun 16 2011 21:21:55 GMT-0400 (EDT)",
    "tLast" : "Thu Jun 16 2011 21:26:49 GMT-0400 (EDT)",
    "now" : "Thu Jun 16 2011 21:27:28 GMT-0400 (EDT)"
}
```

这里有oplog中第一条和最后一条的时间戳，你可以使用`$natural`排序修饰符手工找到这些oplog条目。例如，下面这条查询能获取最后一个条目：`db.oplog.rs.find().sort({$natural:-1}).limit(1)`。

关于复制，还有一件重要的事情，即从节点是如何确定它们在oplog里的位置的。答案在于从节点自己也有一份oplog。这是对主从复制的一项重大改进，因此值得花些时间深究其中的原理。

假设向副本集的主节点发起写操作，接下来会发生什么？写操作先被记录下来，添加到主节点的oplog里。与此同时，所有从节点从主节点复制oplog。因此，当某个从节点准备更新自己时，它做了三件事：首先，查看自己oplog里最后一条的时间戳；其次，查询主节点oplog里所有大于此时间戳的条目；最后，把那些条目添加到自己的oplog里并应用到自己的库里。[①]也就是说，万一发生故障，任何被提升为主节点的从节点都会有一个oplog，其他从节点能以它为复制源进行复制。这项特性对副本集的恢复而言是必需的。

从节点使用长轮询（long polling）立即应用来自主节点oplog的新条目。因此从节点的数据通常都是最新的。由于网络分区或从节点本身进行维护造成数据陈旧时，可以使用从节点oplog里最新的时间戳来监测复制延迟。

2. 停止复制

如果从节点在主节点的oplog里找不到它所同步的点，那么会永久停止复制。发生这种情况时，你会在从节点的日志里看到如下异常：

```
repl: replication data too stale, halting
Fri Jan 28 14:19:27 [replsecondary] caught SyncException
```

回忆一下，oplog是一个固定集合，也就是说集合里的条目最终都会过期。一旦某个从节点没能在主节点的oplog里找到它已经同步的点，就无法再保证这个从节点是主节点的完美副本了。因为修复停止复制的唯一途径是重新完整同步一次主节点的数据，所以要竭尽全力避免这个状态。为此，要监测从节点的延时情况，针对你的写入量要有足够大的oplog。在第10章里能了解到更多与监控有关的内容。接下来我们将讨论如何选择合适的oplog大小。

3. 调整复制OPLOG大小

因为oplog是一个固定集合，所以一旦创建就无法重新设置大小（至少自MongoDB v2.0起是这样的），[②]为此要慎重选择初始oplog大小。

默认的oplog大小会随着环境发生变化。在32位系统上，oplog默认是50 MB，而在64位系统上，oplog会增大到1 GB或空余磁盘空间的5%。[③]对于多数部署环境，空余磁盘空间的5%绰绰有余。对于这种尺寸的oplog，要意识到一旦重写20次，磁盘就可能满了。

因此默认大小并非适用于所有应用程序。如果知道应用程序写入量会很大，在部署之前应该做些测试。配置好复制，然后以生产环境的写入量向主节点发起写操作，像这样对服务器施压起码一小时。完成之后，连接到任意副本集成员上，获取当前复制信息：

```
db.getReplicationInfo()
```

一旦了解了每小时会生成多少oplog，就能决定分配多少oplog空间了。你应该为从节点下线至少八小时做好准备。发生网络故障或类似事件时，要避免任意节点重新同步完整数据，增加oplog大小能为你争取更多时间。

① 开启Journaling日志时，文档会在一个原子事务里被同时写入核心数据文件和oplog。

② 增加固定集合大小的选项已列入计划特性之列，详见https://jira.mongodb.org/browse/SERVER-1864。

③ 如果运行的是OS X，这时oplog将是192 MB。这个值较小，原因是会假设OS X的机器是开发机。

如果要改变默认oplog大小，必须在每个成员节点首次启动时使用mongod的--oplogSize选项，其值的单位是兆。可以像这样启动一个1 GB oplog的mongod实例：

```
mongod --replSet myapp --oplogSize 1024
```

4. “心跳”检测与故障转移

副本集的“心跳”检测有助于选举和故障转移。默认情况下，每个副本集成员每两秒钟ping一次其他所有成员。这样一来，系统可以弄清自己的健康状况。在运行rs.status()时，你可以看到每个节点上次“心跳”检测的时间戳和健康状况（1表示健康，0表示没有应答）。

只要每个节点都保持健康且有应答，副本集就能快乐地工作下去。但如果哪个节点失去了响应，副本集就会采取措施。每个副本集都希望确认无论何时都恰好存在一个主节点。但这仅在大多数节点可见时才有可能。例如，回顾上一节里构建的副本集，如果杀掉从节点，大部分节点依然存在，副本集不会改变状态，只是简单地等待从节点重新上线。如果杀掉主节点，大部分节点依然存在，但没有主节点了。因此从节点被自动提升为主节点。如果碰巧有多个从节点，那么会推选状态最新的从节点提升为主节点。

但还有其他可能的场景。假设从节点和仲裁节点都被杀掉了，只剩下主节点，但没有多数节点——原来的三个节点里只有一个仍处于健康状态。在这种情况下，在主节点的日志里会有如下消息：

```
Tue Feb  1 11:26:38 [rs Manager] replSet can't see a majority of the set,
    relinquishing primary
Tue Feb  1 11:26:38 [rs Manager] replSet relinquishing primary state
Tue Feb  1 11:26:38 [rs Manager] replSet SECONDARY
```

没有了多数节点，主节点会把自己降级为从节点。这让人有点费解，但仔细想想，如果该节点仍然作为主节点的话会发生什么情况？如果出于某些网络原因心跳检测失败了，那么其他节点仍然是在线的。如果仲裁节点和从节点依然健在，并且能看到对方，那么根据多数节点原则，剩下的从节点会变成主节点。要是原来的主节点并未降级，那么你顿时就陷入了不堪一击的局面：副本集中有两个主节点。如果应用程序继续运行，就可能对两个不同的主节点做读写操作，肯定会有不一致，并伴随着奇怪的现象。因此，当主节点看不到多数节点时，必须降级为从节点。

5. 提交与回滚

关于副本集，还有最后一点需要理解，那就是提交的概念。本质上，你可以一直向主节点做写操作，但那些写操作在被复制到大多数节点前，都不会被认为是已提交的。这里所说的已提交是什么意思呢？最好举个例子来做说明。仍以上一节构建的副本集为例，你向主节点发起一系列写操作，出于某些原因（连接问题、从节点为备份而下线、从节点有延迟等）没被复制到从节点。现在假设从节点突然被提升为主节点了，你向新的主节点写数据，而最终老的主节点再次上线，尝试从新的主节点做复制。这里的问题在于老的主节点里有一系列写操作并未出现在新主节点的oplog里。这就会触发回滚。

在回滚时，所有未复制到大多数节点的写操作都会被撤销。也就是说会将它们从从节点的oplog和它们所在的集合里删掉。要是某个从节点里登记了一条删除，那么该节点会从其他副本

里找到被删除的文档并进行恢复。删除集合以及更新文档的情况也是一样的。

相关节点数据路径的rollback子目录中保存了被回滚的写操作。针对每个有回滚写操作的集合，会创建一个单独的BSON文件，文件名里包含了回滚的时间。在需要恢复被回滚的文档时，可以用bsondump工具来查看这些BSON文件，并可以通过mongorestore手工进行恢复。

万一你真的不得不恢复被回滚的数据，你就会意识到应该避免这种情况，幸运的是，从某种程度上来说，这是可以办到的。要是应用程序能容忍额外的写延时，那么就能用上稍后会介绍的写关注，以此确保每次（也可能是每隔几次）写操作都能被复制到大多数节点上。使用写关注，或者更通用一点，监控复制的延迟，能帮助你减轻甚至避免回滚带来的全部问题。

本节中你了解了很多复制的内部细节，可能比预想的还要多，但这些知识迟早会派上用处的。在生产环境里诊断问题时，理解复制是如何工作的会非常有用。

8.2.3　管理

虽然MongoDB提供了自动化功能，但副本集其实还有些潜在的复杂配置选项，接下来，我将详细介绍这些选项。为了让配置简单一些，我也会就哪些选项是能被安全忽略的给出建议。

1. 配置细节

这里我会介绍一些与副本集相关的mongod启动选项，并且描述副本集配置文档的结构。

- 复制选项

先前，你学习了如何使用Shell的rs.initiate()和rs.add()方法初始化副本集。这些方法很方便，但它们隐藏了某些副本集配置选项。这里你将看到如何使用配置文档初始化并修改一个副本集的配置。

配置文档里说明了副本集的配置。要创建配置文档，先为_id添加一个值，要和传给--replSet参数的值保持一致：

```
> config = {_id: "myapp", members: []}
{ "_id" : "myapp", "members" : [ ] }
```

members也是配置文档的一部分，可以像下面这样进行定义：

```
config.members.push({_id: 0, host: 'arete:40000'})
config.members.push({_id: 1, host: 'arete:40001'})
config.members.push({_id: 2, host: 'arete:40002', arbiterOnly: true})
```

你的配置文档看起来应该是这样的：

```
> config
{
  "_id" : "myapp",
  "members" : [
    {
      "_id" : 0,
      "host" : "arete:40000"
    },
    {
      "_id" : 1,
      "host" : "arete:40001"
```

```
    },
    {
      "_id" : 2,
      "host" : "arete:40002",
      "arbiterOnly" : true
    }
  ]
}
```

随后可以把该文档作为rs.initiate()的第一个参数，用这个方法来初始化副本集。

严格说来，该文档由以下部分组成：包含副本集名称的_id字段、members数组（指定了3~12个成员），以及一个可选的子文档（用来指定某些全局设置）。示例副本集里使用了最少的配置参数，外加可选的arbiterOnly设置。

文档中要求有一个_id字段，与副本集的名称相匹配。初始化命令会验证每个成员节点在启动时是否都在--replSet选项里用了这个名称。每个副本集成员都要有一个_id字段，包含从0开始递增的整数，还要有一个host字段，提供主机名和可选的端口。

这里通过rs.initiate()方法初始化了副本集，它是对replSetInitiate命令的简单封装。因此，可以像这样启动副本集：

```
db.runCommand({replSetInitiate: config});
```

config就是一个简单的变量，持有配置文档。一旦初始化完毕，每个集合成员都会在local数据库的system.replset集合里保存一份配置文档的副本。如果查询该集合，你会看到该文档现在有一个版本号了。每次修改副本集的配置，都必须递增这一版本号。

要修改副本集的配置，有一个单独的方法replSetReconfig，它接受一个新的配置文档。新文档可以添加或删除集合成员，还可以修改成员说明和全局配置选项。修改配置文档、增加版本号，以及把它传给replSetReconfig方法，这整个过程很麻烦，所以在Shell里有一些辅助方法来简化这个过程。可以在Shell里输入rs.help()，查看这些辅助方法的列表。注意，你已经用过rs.add()了。

请牢记一点，无论何时，要是重新配置副本集导致重新选举新的主节点，那么所有客户端的连接都会被关闭。这是为了确保客户端不会向从节点发送fire-and-forget风格的写操作。

如果你对通过驱动配置副本集感兴趣的话，可以了解一下rs.add()是如何实现的。在Shell提示符里输入rs.add（不带括号的方法），看看这个方法的工作原理。

● 配置文档选项

到目前为止，我们都局限在最简单的副本集配置文档里。但这些文档还支持很多选项，无论是针对副本集成员还是整个副本集。我们将从成员选项开始进行介绍。注意，你已经见过_id、host和arbiterOnly了，下面还会一起详细介绍其他选项。

- **_id（必填）** 唯一的递增整数，表示成员ID。这些_id值从0开始，每添加一个成员就加1。
- **host（必填）** 保存了成员主机名的字符串，带有可选的端口号。如果提供了端口号，需要用冒号与主机名分隔（例如arete:30000）。如果没有指定端口号，则使用默认端口27017。

- **arbiterOnly**　一个布尔值，true或false，标明该成员是否是仲裁节点。仲裁节点只保存配置数据。它们是轻量级成员，参与主节点选举但本身不参与复制。
- **priority**　一个0~1000的整数，帮助确定该节点被选举为主节点的可能性。在副本集初始化和故障转移时，集合会尝试将优先级最高的节点推选为主节点，只要它的数据是最新的。也有一些场景里，你希望某个节点永远都不会成为主节点（比方说，一个位于从数据中心的灾难恢复节点）。在这些情况中，可以把优先级设置为0。遇到isMaster()命令，带有优先级0的节点会被标记为被动节点，永远都不会被选举为主节点。
- **votes**　所有副本集成员默认都有一票。votes设置让你能给某个单独的成员更多投票。如果要使用该选项，请格外小心。首先，在各个成员的投票数不一致时，很难推测副本集的故障转移行为。其次，绝大多数生产部署环境里，每个成员只有一票的配置工作得都十分理想。因此，要是确定要修改某个指定成员的投票数，一定要经过深思熟虑，并仔细模拟各种故障场景。
- **hidden**　一个布尔值，如果为true，在isMaster命令生成的响应里则不会出现该节点。因为MongoDB驱动依赖于isMaster来获取副本集的拓扑情况,所以隐藏一个成员能避免驱动自动访问它。该设置能同buildIndexes协同使用，使用时必须有slaveDelay。
- **buildIndexes**　一个布尔值，默认为true，确定该成员是否会构建索引。仅当该成员永远不会成为主节点时（那些优先级为0的节点），才能将它设置为false。该选项是为那些只会用作备份的节点设计的。如果备份索引很重要，那么就不要使用它。
- **slaveDelay**　指定从节点要比主节点延迟的秒数。该选项只能用于永远不会成为主节点的节点。所以如果要把slaveDelay设置为大于0的值，务必保证将优先级设置为0。可以通过延迟从节点来抵御某些用户错误。例如，如果有一个延迟30分钟的从节点，管理员不小心删除了数据库，那么在问题扩散之前，你有30分钟做出反应。
- **tags**　包含一个任意键值对集合的文档，通常用来标识成员在某个数据中心或机架的位置。标签被用来指定写关注的粒度和读设置（8.4.9节里会做详细的讨论）。

以上就是针对单个副本集成员的所有选项。还有两个全局副本集配置参数，位于settings键中。在副本集配置文档里，它们是这样的：

```
{
  settings: {
    getLastErrorDefaults: {w: 1},
    getLastErrorModes: {
      multiDC: { dc: 2 }
    }
  }
}
```

- **getLastErrorDefaults**　当客户端不带参数调用getLastError时，默认的参数是由这个文档指定的。要谨慎对待该选项，因为它也可能设置了驱动中getLastError的全局默认值，你可以想象这样一种情况：应用程序开发者调用了getLastError，但他没有意识到管理员在服务器上指定了一个默认值。

关于`getLastError`更详细的信息，可以查看3.2.3节与写关注相关的部分。简单起见，要指定所有写操作都要在500 ms内被复制到至少两个成员上，可以像这样进行配置：`settings: { getLastErrorDefaults: {w: 2, wtimeout: 500} }`。

- **getLastErrorModes** 为`getLastError`命令定义了额外模式的文档。这个特性依赖于副本集标签，详见8.4.4节。

2. 副本集状态

通过`replSetGetStatus`命令能够看到副本集及其成员的状态。要在Shell里调用该命令，可以运行`rs.status()`辅助方法。结果文档标识了现存成员及其各自的状态、正常运行时间和oplog时间。了解副本集成员的状态是非常重要的；在表8-1里可以看到完整的状态值列表。

表8-1 副本集状态

状 态	状态字符串	说 明
0	STARTUP	表示节点正在通过ping与其他节点沟通，分享配置数据
1	PRIMARY	这是主节点。副本集总是有且仅有一个主节点
2	SECONDARY	这是只读的从节点。该节点在故障转移时可能会成为主节点，当且仅当其优先级大于0并且没有被标记为隐藏时
3	RECOVERING	该节点不能用于读写操作。通常会在故障转移或添加新节点后看到这个状态。在恢复时，数据文件通常正在同步中；可以查看正在恢复的节点的日志进行验证
4	FATAL	网络连接仍然建立着，但节点对ping没响应了。节点被标记为FATAL，通常说明托管该节点的机器发生了致命错误
5	STARTUP2	初始数据文件正在同步中
6	UNKNOWN	还在等待建立网络连接
7	ARBITER	该节点是仲裁节点
8	DOWN	该节点早些时候还能访问并正常运行，但现在对“心跳”检测没应答了
9	ROLLBACK	正在进行回滚

当所有节点的状态都是1、2或7，并且至少有一个节点是主节点时，可以认为副本集是稳定且在线的。可以在外部脚本里使用`replSetGetStatus`命令来监控全局状态、复制延时以及正常运行时间，建议在生产环境部署中这样做。[①]

3. 故障转移与恢复

你在示例副本集里已经看过几个故障转移的例子了。这里，我总结一下故障转移的规则，提供几个处理恢复的建议。

当配置中的所有成员都能和其他成员通信时，副本集就能上线了。每个节点默认都有一票投票，那些投票最终会帮助得出投票结果，选出主节点。这意味着只要两个节点（和投票）就能启动副本集了。但初始的投票数还能决定发生故障转移时，什么才能构成多数节点。

让我们假设你配置了一个由三个完整副本（没有仲裁节点）组成的副本集，这也达到了自动

① 除了运行状态命令，还可以通过Web控制台看到有用的信息。第10章讨论了Web控制台，并给出了一些结合副本集的使用示例。

故障转移的推荐最小配置。如果主节点发生故障了，剩下的从节点仍能看到对方，那么就能选出新的主节点。如何做出选择呢？拥有最新oplog（或更高优先级）的从节点会被选为主节点。

- 故障模式与恢复

恢复是在故障后将副本集还原到原始状态的过程。有两大类故障需要处理。第一类包含所谓的无损故障（clean failure），仍然可以认为该节点的数据文件是完好无损的。网络分区（network partition）就是一个例子，若某个节点失去了与其他节点的连接，你只需要等待重新建立连接就行了，被分割开的节点也会重新变为副本集中的成员。还有一个类似的情况，某个节点的mongod进程出于某些原因被终止了，但它可以恢复正常在线状态。[①]同样的，一旦进程重启，它就能重新加入集合了。

第二类故障包含所有明确故障（categorical failure），某个节点的数据文件不存在或者必须假设已经损坏。非正常关闭mongod进程，又没有开启Journaling日志，以及硬盘崩溃都属于此类故障。恢复明确故障节点的唯一途径就是重新同步或利用最近的备份完全替换数据文件，让我们轮流看下这两种策略。

要完全重新同步，在故障节点上的某个空数据目录里启动一个mongod。只要主机名和端口号没有改变，新的mongod会重新加入副本集，随后重新同步全部现有数据。如果主机名或者端口号有变化，那么在mongod重新上线后，你还需要重新配置副本集。举个例子，假设节点arete:40001的数据无法恢复，你在foobar:40000启动了一个新节点。你可以重新配置副本集，只需抓取配置文档，修改第二个节点的host属性，随后将其传给rs.reconfig()方法：

```
> use local
> config = db.system.replset.findOne()
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
    {
      "_id" : 0,
      "host" : "arete:40000"
    },
    {
      "_id" : 1,
      "host" : "arete:40001"
    },
    {
      "_id" : 2,
      "host" : "arete:40002"
    }
  ]
}
> config.members[1].host = "foobar:40000"
foobar:40000
> rs.reconfig(config)
```

① 举例来说，如果MongoDB是正常关闭的，那你肯定知道数据文件是好的。或者，如果使用了Journaling日志，不管是如何结束的，MongoDB实例都能恢复。

现在副本集可以识别新节点了，而新节点应该能从现有节点同步数据了。

除了通过完全重新同步进行恢复，还可以通过最近的备份进行恢复。通常都会使用某个从节点来进行备份[①]，方法是制作数据文件的快照并离线存储。仅当备份中的oplog不比当前副本集成员的oplog旧时，才能通过备份进行恢复。也就是说，备份的oplog里的最新操作必须仍存在于线上oplog里。可以用`db.getReplicationInfo()`提供的信息立即确定情况是否如此。在进行恢复时，不要忘记考虑还原备份所需的时间。要是备份里最新的oplog条目在从备份复制到新机器的过程有可能变旧，那么最好还是进行完全重新同步吧。

但通过备份进行恢复速度更快，部分原因是不用从零开始重新构建索引。要从备份进行恢复，将备份的数据文件复制到`mongod`的数据路径里。应该会自动开始重新同步的，你可以检查日志或者运行`rs.status()`进行验证。

4. 部署策略

你已经知道了副本集最多可以包含12个节点，看过了一组令人眼花缭乱的配置选项表格，以及故障转移与恢复所要考虑的内容。配置副本集的方式有很多，但在本节中，我只会讨论那些适用于大多数情况的配置方式。

提供自动故障转移的最小副本集配置就是先前所构建的那个，包含两个副本和一个仲裁节点。在生产环境中，仲裁节点可以运行在应用服务器上，而副本则运行于自己的机器上。对于多数生产环境中的应用而言，这种配置既经济又高效。

但是对于那些对正常运行时间有严格要求的应用程序而言，副本集中需要包含三个完整的副本。那个额外的副本能带来什么好处呢？请想象这样一个场景：一个节点彻底损坏了。在恢复损坏的节点时，你还有两个正常的节点可用。只要第三个节点在线并正在恢复（这可能需要几个小时），副本集仍能自动故障转移到拥有最新数据的节点上。

一些应用程序要求有两个数据中心来做冗余，三个成员的副本集在这种情况下仍然适用。技巧在于让其中一个数据中心仅用于灾难恢复。图8-2就是一个例子。其中，主数据中心运行了副本集的主节点和一个从节点，备用数据中心里的从节点作为被动节点（优先级为`0`）。

在这个配置中，副本集的主节点始终是数据中心A里两个节点的其中之一。你可以在损失任意一个节点或者任意一个数据中心的情况下，保持应用程序在线。故障转移通常都是自动的，除非数据中心A的节点都发生了故障。同时损失两个节点的情况很少见，通常表现为数据中心A完全故障或者网络分区。要迅速恢复，可以关闭数据中心B里的节点，不带`--replSet`参数进行重启。除此之外，还可以在数据中心B里启动两个新节点，随后强制进行副本集重新配置。照道理不该在大多数节点无法访问时重新配置副本集，但在紧急情况下可以使用`force`选项这么做。例如，假设定义了一个新的配置文档config，可以像下面这样强制进行重新配置：

```
> rs.reconfig(config, {force: true})
```

和所有生产系统一样，测试是关键，请确保在类似于生产环境的预发布环境中对所有典型故障转移和恢复场景进行测试。了解副本集在这些故障情况下会有何表现，这会让你在发生紧急情

① 第10章会详细讨论备份。

况时更从容不迫、处乱不惊。

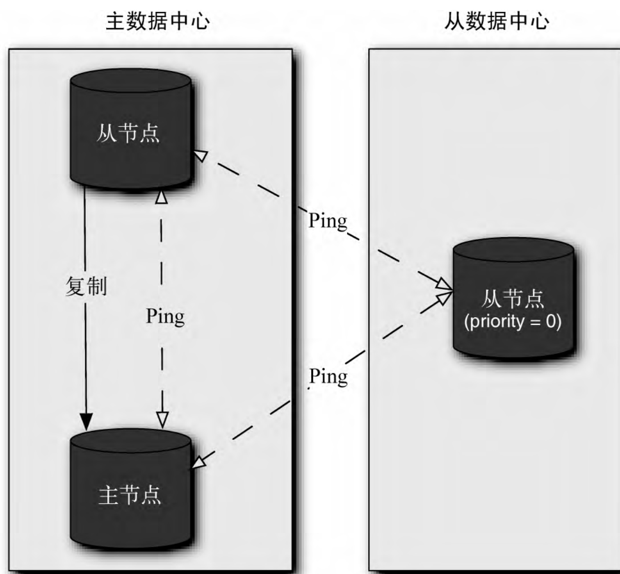


图8-2　成员分布在两个数据中心的三节点副本集

8.3　主从复制

主从复制是MongoDB最初使用的复制范式。这种复制易于配置，能支持任意数量的从节点。但是出于一些原因，我们不再推荐在生产部署中使用主从复制了。首先，故障转移完全是人工操作的。如果主节点发生故障，管理员必须关闭某个从节点，把它重启为主节点，随后应用程序必须重新配置以指向新的主节点。其次，恢复很困难。因为oplog仅存在于主节点上，发生故障后要求在新的主节点上创建新的oplog。这意味着在发生故障时，其他现有节点都需要从新的主节点上重新进行同步。

简而言之，没有什么有说服力的理由使用主从复制。副本集才是正途，你应该使用这种复制方式。

8.4　驱动与复制

如果正在构建应用程序，并且使用了MongoDB的复制功能，那么你需要了解三个特定于应用的话题。第一个主题与连接和故障转移有关；随后是写关注允许你决定在应用程序继续下一步之前写操作的复制程度；最后是读扩展，允许应用程序将读请求分布在多个副本之间。我会依次讨论这些话题。

8.4.1　连接与故障转移

MongoDB的驱动提供了一套相对统一的界面来连接副本集。

1. 单节点连接

你总是可以连接到副本集里的单个节点上。连接到副本集的主节点和连接到普通的单机节点（正如我们全书中的例子那样）没有什么区别。这两种情况下，驱动都会初始化一个TCP套接字连接，运行`isMaster`命令。这条命令会返回如下文档：

```
{ "ismaster" : true, "maxBsonObjectSize" : 16777216, "ok" : 1 }
```

对于驱动而言，最重要的是该节点的`isMaster`字段是设置为`true`的，这表明指定节点可以是单机、主从复制里的主节点或者副本集的主节点。[1]在所有这些情况里，节点都能写入，驱动的用户能执行各种CRUD操作。

但在直接连接到副本集的从节点时，必须标明你*知道*自己正在连接从节点（至少对大多数驱动而言需要如此）。在Ruby驱动里，你可以带上`:slave_ok`参数。于是，直接连接本章之前创建的第一个从节点的Ruby代码是这样的：

```
@con = Mongo::Connection.new('arete', 40001, :slave_ok => true)
```

没有`:slave_ok`参数，驱动会抛出一个异常，指出无法连接到主节点。这个检查是为了避免无意中向从节点进行写操作。虽然这种写操作会被服务器拒绝，但你看不到任何异常，除非使用安全模式进行操作。

MongoDB假设你通常都会连接主节点；`:slave_ok`参数可以用来作为一道强制的健康检查。

2. 副本集连接

虽然你能单独连接副本集的各个成员，但一般都会希望连接整个副本集。这能让驱动确定哪个节点是主节点，并在故障转移时重新连接新的主节点。

大多数官方支持的驱动都提供了连接副本集的方法。在Ruby驱动里，可以创建一个`ReplSetConnection`实例，传入*种子节点*（seed node）列表：

```
Mongo::ReplSetConnection.new(['arete', 40000], ['arete', 40001])
```

驱动内部会尝试连接各个种子节点，并调用`isMaster`命令，该命令会返回一些重要的集合细节：

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
    "arete:40000",
    "arete:40001"
  ],
```

8

① `isMaster`命令还会返回该版本服务器的最大BSON对象大小。随后，驱动会在插入BSON对象前验证所有这些对象是否满足此限制。

```
  "arbiters" : [
    "arete:40002"
  ],
  "maxBsonObjectSize" : 16777216,
  "ok" : 1
}
```

一旦某个种子节点返回如上信息，驱动就拿到它需要的所有信息了。现在它能连接主节点，再次验证该成员依然是主节点，然后允许用户通过该节点进行读写操作。响应对象还允许驱动缓存剩余的从节点和仲裁节点的地址。如果主节点上的操作失败，那么后续的请求中，驱动都会尝试连接剩余的某个节点，直到它能重新连上主节点。

请牢记一点，虽然副本集的故障转移是自动的，但驱动不会隐藏发生故障这一事实。处理过程大致是这样的：首先，主节点发生故障或者发生了新的选举。后续的请求会显示套接字连接已断开，驱动就抛出一个连接异常，关闭那些打开的连接数据库的套接字。随后由应用程序开发者来决定该怎么办，这一决定依赖于要执行的操作和应用程序的特定需求。

请记住，在处理后续请求时，驱动会自动尝试重新连接，让我们想象几个场景。首先，假设你只向数据库发送读请求。在这种情况下，重试失败的读操作不会产生危害，因为它不会改变数据库的状态。但是，再假设通常还会向数据库发送写请求。之前提到过多次，无论是否开启安全模式，你都能写数据库。在安全模式下，驱动在每次写入后会追加一次`getlasterror`命令调用，这能确保写操作已安全到达并向应用程序报告各种服务器错误。不使用安全模式时，驱动只是简单地向TCP套接字做写操作。

如果应用在没有使用安全模式时执行写入并发生故障转移，就会产生不确定的状态。最近向服务器做了多少写操作？有多少是丢失在套接字缓存里的？向TCP套接字做写操作的不确定性让你无法回答这些问题。这个问题有多严重取决于应用程序。对日志而言，不安全的写入也许是可接受的，因为丢失几条日志不会影响日志的全貌；但对于用户创建的数据，这就是一场灾难。

开启安全模式后，只有最后一次的写操作会有问题；可能它已经到服务器了，也可能没有。有时可能会重试，也可能会抛出一个应用程序错误。驱动始终会抛出一个异常；然后，开发者能够决定如何处理这些异常。

不管什么情况，重试一个操作都会让驱动尝试重新连接副本集。由于不同的驱动在副本集的连接行为上稍有不同，你应该查看驱动的文档了解详细信息。

8.4.2　写关注

现在情况已经很明朗了，默认运行安全模式对于大多数应用程序都是合理的，因为能够知道写操作正确无误地到达主节点是很重要的。但人们通常都会希望有更高级别的保证，写关注就能做到这点，它允许开发者指定应用程序执行后续操作前写操作应该被复制的范围。严格说来，你是通过`getlasterror`命令的两个参数来控制写关注的：`w`和`wtimeout`。

第一个参数`w`，接受的值通常都是最近的写操作应该被复制到的服务器的总数；第二个参数

是超时，如果写操作在指定毫秒内无法复制，该命令就会返回一个错误。

例如，如果你希望写操作至少要复制到一台服务器上，可以将w指定为2。如果希望在500 ms内无法完成该复制就超时，可以将wtimeout指定为500。请注意，如果不指定wtimeout的值，而复制又出于某些原因一直没有发生，那么该操作会一直阻塞下去。

在使用驱动时，不是通过显式调用getLastError开启写关注的，而是创建一个写关注对象，或者设置合适的安全模式选项；这依赖于特定驱动的API。[①]在Ruby里可以像这样为一个操作设置写关注：

```
@collection.insert(doc, :safe => {:w => 2, :wtimeout => 200})
```

有时，你只是想确保写操作被复制到了大部分可用节点上，这时可以简单地将w值设置为majority：

```
@collection.insert(doc, :safe => {:w => "majority"})
```

还有更高级的选项。举例来说，如果已经开启了Journaling日志，还可以通过j选项强制让Journaling日志同步到磁盘上：

```
@collection.insert(doc, :safe => {:w => 2, :j => true})
```

很多驱动还支持为指定连接或数据库设置写关注的默认值。要了解如何在具体场景中设置写关注，请查看所用驱动的文档。附录D中能找到更多语言的例子。

写关注既能用于副本集，也能用于主从复制。如果查看local数据库，你会看到两个集合，从节点上的me和主节点上的slaves，它们就是用来实现写关注的。每当从节点从主节点同步数据时，主节点都会在slaves集合里记录下应用到从节点上的最新oplog条目。因此，主节点总是能知道每个从节点复制了什么东西，可以准确地响应带getlasterror命令的写请求。

请记住，使用写关注时w值大于1会引入额外的延时。可配置的写关注让你能够在速度和持久性之间做出权衡。如果使用了Journaling日志，那么w等于1就已经能满足大多数应用程序的需要了。另一方面，对于日志或分析型的应用程序，你可能会选择同时禁用Journaling日志和写关注，仅依靠复制来保证持久性，这在发生故障时可能会丢失一些写入的数据。请仔细考虑这些因素，在设计应用程序时测试不同的场景。

8.4.3 读扩展

经复制的数据库能很好地适用于读扩展。如果单台服务器无法承担应用程序的读负载，那么可以将查询路由到更多的副本上。大多数驱动都内置了将查询发送到从节点的功能。在Ruby驱动中，ReplSetConnection构造方法的一个选项就提供了对该功能的支持：

```
Mongo::ReplSetConnection.new(['arete', 40000],
          ['arete', 40001], :read => :secondary )
```

当:read参数被设置为:secondary时，连接对象会随机选择一个附近的从节点读取数据。

① 附录D中包含在Java、PHP和C++里设置写关注的例子。

其他驱动可以通过设置slaveOk选项进行配置，读取从节点数据。当使用Java驱动连接副本集时，将slaveOk设置为true将以每个线程为基础，开启从节点的负载均衡。驱动中的负载均衡实现是为普通应用设计的，因此可能无法适用于所有应用。遇到这种情况时，用户通常会定制自己的负载均衡实现。同样的，请查看你的驱动文档了解更多细节。

很多MongoDB用户在生产环境中通过复制进行扩展。但是，有三种情况复制无法应对。第一种情况与所需的服务器数量有关，自MongoDB v2.0起，副本集最多支持12个成员，其中7个可以投票。如果需要更多副本来做扩展，可以使用主从复制。但如果既不想牺牲自动故障转移，又要超过副本集的成员上限，那就需要迁移到分片集群上了。

第二种情况涉及那些写负载较高的应用程序。正如本章开篇时所说的那样，从节点必须跟上这个写负载。向那些满负荷做写操作的从节点发送读请求可能会妨碍复制。

第三种副本扩展无法处理的情况是一致性读。因为复制是异步的，副本无法始终反映主节点最新的写操作。因此，如果应用程序任意地从多个从节点读取数据，那么呈现给最终用户的内容不能始终保证是完全一致的。对于那些主要用来显示内容的应用程序而言，这几乎从来都不是问题。但对于其他应用而言，用户是在主动操作数据，这就要求一致性读。在这些情况下，你有两个选择。第一是将那些需要一致性读的应用程序部分从那些不需要的部分里分离出来。前者总是从主节点读取数据，后者可以从多个从节点读取数据。当这种策略太复杂或者无法扩展时，就该采取分片策略。[①]

8.4.4　标签

如果正在使用写关注或者读扩展，你可能会想要更细粒度地进行控制，控制哪个从节点接收写或读请求。例如，假设部署了一个五节点副本集，跨两个数据中心：NY和FR。主数据中心NY包含三个节点，从数据中心FR包含剩下的两个节点。假设希望通过写关注阻塞请求，直到写操作被复制到数据中心FR的至少一个节点上。以目前你所了解的写关注知识来看，没有什么好办法实现这一需求。w值为majority是没用的，因为这会被翻译成值3，最可能的情况是NY里的三个节点先发出响应。也可以将值设置为4，但如果每个数据中心各损失一个节点，那这种方法也会有问题。

副本集标签可以解决这个问题，它允许针对带有特定标签的副本集成员定义特殊的写关注模式。要知道这是如何实现的，先要了解如何为副本集成员打标签。在配置文档里，每个成员都有一个名为tags的键指向一个包含键值对的对象。下面就是一个例子：

```
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
    {
      "_id" : 0,
      "host" : "ny1.myapp.com:30000",
```

① 注意，要从分片集群中获得一致性读，必须始终读取每个分片的主节点，而且必须发起安全写操作。

```
      "tags": { "dc": "NY", "rackNY": "A" }
    },
    {
      "_id" : 1,
      "host" : "ny2.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "A" }
    },
    {
      "_id" : 2,
      "host" : "ny3.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "B" }
    },
    {
      "_id" : 3,
      "host" : "fr1.myapp.com:30000",
      "tags": { "dc": "FR", "rackFR": "A" }
    },
    {
      "_id" : 4,
      "host" : "fr2.myapp.com:30000",
      "tags": { "dc": "FR", "rackFR": "B" }
    }
  ],

  settings: {
    getLastErrorModes: {
      multiDC: { dc: 2 } },
      multiRack: { rackNY: 2 } },
    }
  }
}
```

这个带标签的配置文档适用于之前假设的跨两个数据中心的副本集。请注意，每个成员的标签文档有两个键值对：第一个标识了数据中心，第二个是指定节点服务器所在机架的名称。请记住，这里使用的名称是完全任意的，而且仅在本应用程序的上下文中有意义；你可以在标签文档中放置任何东西。重要的是如何使用它。

这时getLastErrorModes该登场了。它们允许为getLastError命令定义模式，这些模式实现了特殊的写关注要求。在本例中，你定义了两个模式，第一个是multiDC，定义为{"dc":2}，表示写操作应该复制到至少有两个不同dc值的节点上。如果这时检查标签，你会看到它能确保写操作已经传播到了两个数据中心。第二个模式规定了至少要有两个NY的机架接收到了写操作。这同样也能通过标签加以实现。

一般来说，一个getLastErrorModes条目包含一个文档，其中有一或多个键（本例中是dc和rackNY），它们的值是整数。这些整数表示某个键的不同标签值数量，在getLastError命令成功完成时必须满足这些值。一旦定义好了这些模式，就能在应用程序里将其用作w的值。例如，在Ruby中使用第一个模式，如下：

8

```
@collection.insert(doc, :safe => {:w => "multiDC"})
```

除了能让写关注更加精细，标签还能提供更粒度化的控制，决定哪个副本用于读扩展。可惜在本书编写时，针对标签进行读操作的语义尚未定义或实现在官方MongoDB驱动里。要了解最新进展，请查看Ruby驱动的JIRA问题单，参见https://jira.mongodb.org/browse/RUBY-326。

8.5　小结

综上所述，复制功能十分有用，而且在大多数部署环境中都是必不可少的。MongoDB的复制比较简单，配置通常也很方便。但在需要备份和故障转移时，还是隐藏了一定复杂性的。针对这些复杂的情况，希望经验和本章内容能给你带来一定帮助。

第9章 分　　片

本章内容

- ❑ 分片的概念
- ❑ 配置并加载示例分片集群
- ❑ 管理与故障转移

MongoDB在设计之初就支持分片，这是一个宏伟的目标，因为要构建一个支持自动基于范围进行分区和负载均衡，并且没有单点故障的系统是非常困难的。对生产级分片的支持最早出现在2010年8月发布的MongoDB v1.6里，自那以后，分片子系统经历了无数的改进。高效地分片能让用户在节点间均匀分布大量数据，并按需增加容量。本章，我会介绍MongoDB引以为荣的分片机制。

首先是分片的概述，讨论什么是分片，为什么它这么重要，以及在MongoDB里它是如何实现的。虽然这能让你了解基本的分片知识，但在动手配置自己的分片集群前，你都无法完全掌握它。而这正是你在第二节里要做的：构建一个示例集群，托管一个与Google Docs类似的应用程序的大量数据。我们随后会讨论一些分片机制，描述查询与索引是如何在分片里工作的。我们还会了解到如何选择分片键，这点至关重要。本章结尾处，我将给出很多在生产环境中运行分片的具体建议。

分片是很复杂的，要想学好本章的内容，你应该运行其中的示例。在一台机器上运行示例集群应该不成问题；一旦成功运行，你就可以动手进行试验了。要想理解作为分布式系统的MongoDB，没有什么比拥有一个分片集群更好的了。

9.1　分片概述

在你构建第一个分片集群之前，有必要理解什么是分片以及为什么有时它能适用。为什么分片很重要？对此的说明是整个MongoDB项目的核心选择理由之一。一旦理解了为什么分片如此重要，你将欣喜地了解到组成分片集群的核心组件，还有构成MongoDB分片机制的关键概念。

9.1.1 何谓分片

到目前为止，你都是把MongoDB当做一台服务器在用，每个`mongod`实例都包含应用程序数据的完整副本。就算使用了复制，每个副本也都完整克隆了其他副本的数据。对于大多数应用程序而言，在一台服务器上保存完整数据集是完全可以接受的。但随着数据量的增长，以及应用程序对读写吞吐量的要求越来越高，普通服务器渐渐显得捉襟见肘了。尤其是这些服务器可能无法分配足够的内存，或者没有足够的CPU核数来有效处理工作负荷。除此之外，随着数据量的增长，要在一块磁盘或一组RAID阵列上保存和管理备份如此大规模的数据集也变得不太现实了。如果还想继续使用普通硬件或者虚拟硬件来托管数据库，那么针对这类问题的解决方案就是将数据库分布在多台服务器上。这种方法称为分片。

为数众多的Web应用程序，知名的有Flickr和LiveJournal，都实现了手动分片，将负载分布到多台MySQL数据库上。在这些实现中，分片逻辑都寄生于应用程序之中。要明白这是如何实现的，想象一下，假设你有很多用户，需要将`Users`表分布到多台数据库服务器上。你可以指定一台数据库作为元数据库。这台数据库包含每个用户ID（或者用户ID范围）到指定分片映射关系的元数据。因此，要查询一个用户实际涉及两次查询：第一次查询访问元数据库以获得用户的分片位置，第二次查询直接访问包含用户数据的分片。

对于这些Web应用程序而言，手动分片解决了负载问题，但其实现并非无懈可击。最明显的问题是迁移数据非常困难。如果单个分片负载过重，将其中的数据迁移到其他分片的过程完全是手动的。手动分片的第二个问题在于要编写可靠的应用程序代码对读写请求进行路由，并且将数据库作为一个整体进行管理，这也是非常困难的。最近出现了一些管理手动分片的框架，最著名的就是Twitter的Gizzard（详见http://mng.bz/4qvd）。

但正如那些手动分片数据库的人所说的，要把事情做好并非易事。MongoDB中有一大块工作就是为了解决该问题。因为分片是MongoDB的核心内容，所以用户无需担心在需要水平扩展时要自己设计外置分片框架。在处理困难的跨分片数据均衡问题时，这点尤为重要。这些代码并非那种大多数人能在一个周末里写出来的东西。

也许最值得一提的是MongoDB在设计时为应用程序提供了统一的接口，无论是在分片前，还是在分片后。也就是说，在数据库需要转换为分片架构时，应用程序代码几乎无需改动。

现在你应该对自动分片背后的逻辑有点感觉了。在详细描述MongoDB的分片过程前，让我们停下脚步，回答另一个摆在面前的问题：何时需要分片？

何时分片

这个问题的答案比你想的要简单得多。我们之前已经说过把索引和工作数据集放在内存里是很重要的，这也是分片的主要原因。如果应用程序的数据集持续无限增长，那么迟早有一天，内存会容纳不下这些数据。如果你正使用亚马逊的EC2，那么这个阈值是68 GB，因为这是本书编写时EC2最大的实例所能提供的内存总数。或者，你可以运行自己的硬件，并使用远高于68 GB的内存，这样便能延后一段时间再做分片。但没有哪台机器的内存是无限的，因此你早晚都会用到分片。

不可否认，还有一些其他的应对措施。举例来说，如果你有自己的硬件，而且可以将所有数据都保存在固态硬盘上（它的成本越来越能为人所接受了），那么可以增加数据内存比，而不会为性能带来负面影响。还有一种情况，工作集是总数据量中的一部分，这时可以使用相对较小的内存。另一方面，如果有特殊的写负载要求，那么可以在数据达到内存大小之前先进行适当的分片，原因是需要将负载分到多台机器上，以便能够获得想要的写吞吐量。

无论哪种情况，对现有系统进行分片的决定都要基于以下几点——磁盘活动、系统负载以及最重要的工作集大小与可用内存的比例。

9.1.2　分片的工作原理

要理解分片是如何工作的，你需要了解构成分片集群的组件，理解协调那些组件的软件进程，这就是接下来的主题。

1. 分片组件

分片集群由分片、`mongos`路由器和配置服务器组成。我们所要讨论的组件如图9-1所示。

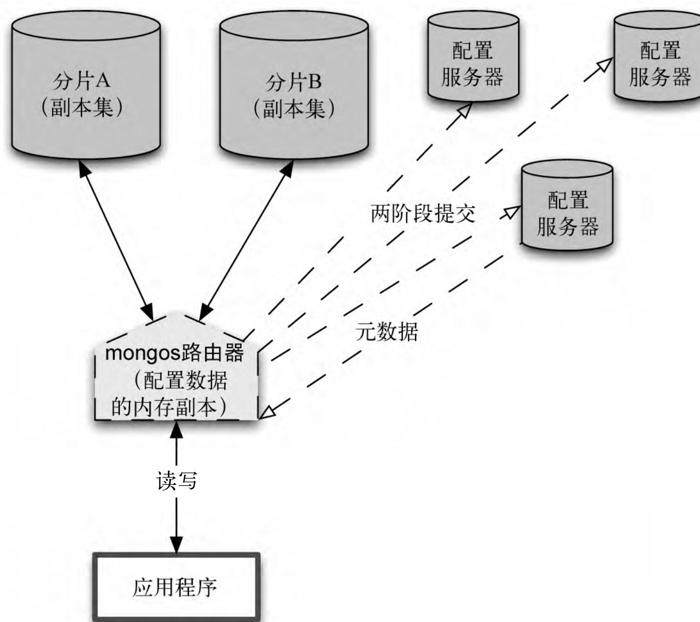


图9-1　MongoDB分片集群中的组件

- 分片

MongoDB分片集群将数据分布在一个或多个分片上。每个分片都部署成一个MongoDB副本集，该副本集保存了集群整体数据的一部分。因为每个分片都是一个副本集，所以它们拥有自己的复制机制，能够自动进行故障转移。你可以直接连接单个分片，就像连接单独的副本集那样。

但是，如果连接的副本集是分片集群的一部分，那么你只能看到部分数据。

- mongos路由器

如果每个分片都包含部分集群数据，那么还需要一个接口连接整个集群，这就是mongos。mongos进程是一个路由器，将所有的读写请求指引到合适的分片上。如此一来，mongos为客户端提供了一个合理的系统视图。

mongos进程是轻量级且非持久化的。它们通常运行于与应用服务器相同的机器上，确保对任意分片的请求只经过一次网络跳转。换言之，应用程序连接本地的mongos，而mongos管理了指向单独分片的连接。

- 配置服务器

如果mongos进程是非持久化的，那么必须有地方能持久保存集群的公认状态；这就是配置服务器的工作，其中持久化了分片集群的元数据，该数据包括：全局集群配置；每个数据库、集合和特定范围数据的位置；一份变更记录，保存了数据在分片之间进行迁移的历史信息。

配置服务器中保存的元数据是某些特定功能和集群维护时的重中之重。举例来说，每次有mongos进程启动，它都会从配置服务器获取一份元数据的副本。没有这些数据，就无法获得一致的分片集群视图。该数据的重要性对配置服务器的设计和部署策略也有影响。

查看图9-1，你会看到三个配置服务器，但它们并不是以副本集的形式部署的。它们比异步复制要求更严格；mongos进程向配置服务器写入时，会使用两阶段提交。这能保证配置服务器之间的一致性。在各种生产环境的分片部署中，必须运行三个配置服务器，这些服务器都必须部署在独立的机器上以实现冗余。[①]

现在你了解了分片集群的构成，但也许还对分片机制本身心存疑惑。数据究竟是如何分布的？接下来我会介绍一些核心分片操作，对此做出解释。

2. 核心分片操作

MongoDB分片集群在两个级别上分布数据。较粗的是以数据库为粒度的，在集群里新建数据库时，每个数据库都会被分配到不同的分片里。如果不进行什么别的设置，数据库以及其中的集合永远都会在创建它的分片里。

因为大多数应用程序都会把所有的数据保存在一个物理数据库里，因此这种分布方式带来的帮助不大。你需要更细粒度的分布方式，集合的粒度刚好能满足要求。MongoDB的分片是专门为了将单独的集合分布在多个分片里而设计的。要更好地理解这点，让我们一起想象一下在真实的应用程序里这是如何工作的。

假设你正在构建一套基于云的办公套件，用于管理电子表格，并且要将所有的数据都保存在MongoDB里。[②] 用户可以随心所欲地创建大量文档，每个文档都会保存为单独的MongoDB文档，放在一个spreadsheets集合里。随着时间的流逝，假设你的应用程序发展到了拥有100万用户。

① 你也可以运行单个配置服务器，但这只能作为简单测试分片的一种手段。在生产环境里只用一台配置服务器就好比乘坐单引擎喷气飞机横跨大西洋：它能带你飞过去，但是一旦失去一个引擎，你就得游泳了。

② 可以参考一下Google Docs之类的产品，Google Docs允许用户创建电子表格和演示幻灯片。

现在再想想那两个主要集合：users和spreadsheets。users集合还比较容易处理。就算有100万用户，每个用户文档1 KB，整个集合大概也就1 GB，一台机器就能搞定了。但spreadsheets集合就大不一样了，假设每个用户平均拥有50张电子表格，平均大小是50 KB，那么我们所谈论的就是1 TB的spreadsheets集合。要是这个应用程序的活跃度很高，你会希望将数据放在内存里。要将数据放在内存里并且分布读写负载，你就必须将集合分片。这时分片就该登场了。

● 分片一个集合

MongoDB的分片是基于范围的。也就是说分片集合里的每个文档都必须落在指定键的某个值范围里。MongoDB使用所谓的分片键（shard key）让每个文档在这些范围里找到自己的位置。[①]从假想的电子表格管理应用程序里拿出一个示例文档，这样你能更好地理解分片键：

```
{
  _id: ObjectId("4d6e9b89b600c2c196442c21")
  filename: "spreadsheet-1",
  updated_at: ISODate("2011-03-02T19:22:54.845Z"),
  username: "banks",
  data: "raw document data"
}
```

在对该集合进行分片时，必须将其中的一个或多个字段声明为分片键。如果选择_id，那么文档会基于对象ID的范围进行分布。但是，出于一些原因（稍后会做说明的），你要基于username和_id声明一个复合分片键；因此，这些范围通常会表示为一系列用户名。

现在你需要理解块（chunk）的概念，它是位于一个分片中的一段连续的分片键范围。举例来说，可以假设docs集合分布在两个分片A和B上，它被分成了如表9-1所示的多个块。每个块的范围都由起始值和终止值来标识。

表9-1 块与分片

起始值	终止值	分片
-∞	abbot	B
abbot	dayton	A
dayton	harris	B
harris	norris	A
norris	∞	B

粗略扫一眼表9-1，你会发现块的一个重要的、有些违反直觉的属性：虽然每个单独的块都表示一段连续范围的数据，但这些块能出现在任意分片上。

关于块，第二个要点是它们是种逻辑上的东西，而非物理上的。换言之，块并不表示磁盘上连续的文档。从一定程度上来说，如果一个从*harris*开始到*norris*结束的块存在于分片A上，那么就认为可以在分片A的docs集合里找到分片键落在这个范围里的文档。这和集合里那些文档的排列没有任何必然关系。

① 其他的分布式数据库里可能使用分区键（partition key）或分布键（distribution key）来代替分片键这个术语。

- 拆分与迁移

分片机制的重点是块的拆分（splitting）与迁移（migration）。

首先，考虑一下块拆分的思想。在初始化分片集群时，只存在一个块，这个块的范围涵盖了整个分片集合。那该如何发展到有多个块的分片集群呢？答案就是块大小达到某个阈值时会对块进行拆分。默认的块的最大块尺寸是64 MB或者100 000个文档，先达到哪个标准就以哪个为准。在向新的分片集群添加数据时，原始的块最终会达到某个阈值，触发块的拆分。这是一个简单的操作，基本就是把原来的范围一分为二，这样就有了两个块，每个块都有相同数量的文档。

请注意，块的拆分是个逻辑操作。当MongoDB进行块拆分时，它只是修改块的元数据就能让一个块变成两个。因此，拆分一个块并不影响分片集合里文档的物理顺序。也就是说拆分既简单又快捷。

你可以回想一下，设计分片系统时最大的一个困难就是保证数据始终均匀分布。MongoDB的分片集群是通过在分片中移动块来实现均衡的。我们称之为迁移，这是一个真实的物理操作。

迁移是由名为均衡器（balancer）的软件进程管理的，它的任务就是确保数据在各个分片中保持均匀分布。通过跟踪各分片上块的数量，就能实现这个功能。虽然均衡的触发会随总数据量的不同而变化，但是通常来说，当集群中拥有块最多的分片与拥有块最少的分片的块数差大于8时，均衡器就会发起一次均衡处理。在均衡过程中，块会从块较多的分片迁移到块较少的分片上，直到两个分片的块数大致相等为止。

如果现在你还不太理解，不用担心。下一节里我会通过一个示例集群来演示分片，通过实践来进一步阐述分片键和块的概念。

9.2　示例分片集群

理解分片的最佳途径就是了解它实际是怎么工作的。幸运的是可以在一台机器上配置分片集群，接下来我们就会这么做，[①] 还会模拟上一节里提到的基于云的电子表格应用程序的行为。在此过程中，我们会仔细查看全局分片配置，通过第一手资料来了解数据是如何基于分片键进行分区的。

9.2.1　配置

配置分片集群有两个步骤。第一步，启动所有需要的mongod和mongos进程。第二步，也是比较简单的一步，发出一系列命令来初始化集群。你将构建的分片集群由两个分片和三个配置服务器组成，另外还要启动一个mongos与集群通信。你要启动的全部进程如图9-2所示，括号里是它们的端口号。

① 为了进行测试，你可以在单台机器上运行各个mongod和mongos进程。在本章后续的内容里，我们会看到生产环境下的分片配置，以及一套切实可行的分片部署所需的最小服务器数量。

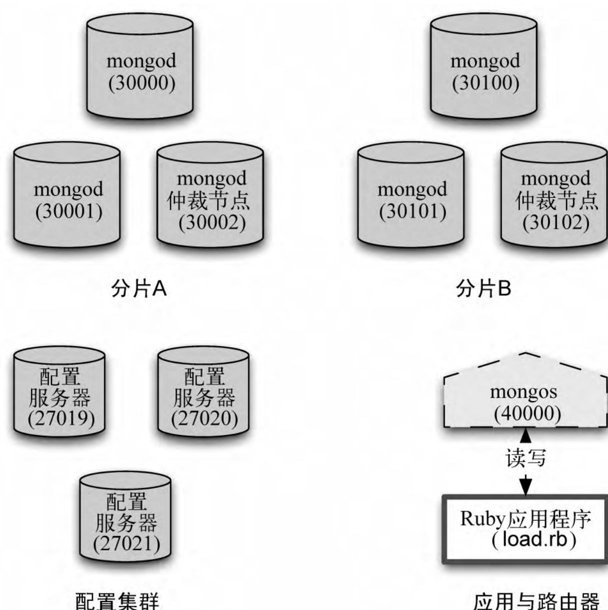


图9-2　由示例分片集群构成的进程全貌

你要运行一堆命令来启动集群，因此如果觉得自己一叶障目，不见泰山，不妨回头看看这张图。

1. 启动分片组件

让我们开始为两个副本集创建数据目录吧，它们将成为分片的一部分。

```
$ mkdir /data/rs-a-1
$ mkdir /data/rs-a-2
$ mkdir /data/rs-a-3
$ mkdir /data/rs-b-1
$ mkdir /data/rs-b-2
$ mkdir /data/rs-b-3
```

接下来，启动每个`mongod`进程。因为要运行很多进程，所以可以使用`--fork`选项，让它们运行在后台。[①]以下是启动第一个副本集的命令。

```
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-1 \
  --port 30000 --logpath /data/rs-a-1.log --fork --nojournal
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-2 \
  --port 30001 --logpath /data/rs-a-2.log --fork --nojournal
$ mongod --shardsvr --replSet shard-a  --dbpath /data/rs-a-3 \
  --port 30002 --logpath /data/rs-a-3.log --fork --nojournal
```

以下是启动第二个副本集的命令：

① 注意，如果运行在Windows上，`fork`是没用的。因为必须打开新终端窗口来运行每个进程，最好把`logpath`选项也忽略了。

```
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-1 \
  --port 30100 --logpath /data/rs-b-1.log --fork --nojournal
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-2 \
  --port 30101 --logpath /data/rs-b-2.log --fork --nojournal
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-3 \
  --port 30102 --logpath /data/rs-b-3.log --fork --nojournal
```

如往常一样，现在要初始化这些副本集了。单独连上每个副本集，运行`rs.initiate()`，随后添加剩余的节点。第一个副本集上的命令是这样的：[①]

```
$ mongo arete:30000
> rs.initiate()
```

大概一分钟之后，初始节点就变成主节点了，随后就能添加剩余的节点了：

```
> rs.add("arete:30001")
> rs.add("arete:30002", {arbiterOnly: true})
```

初始化第二个副本集的方法与之类似。在运行`rs.initiate()`后等待一分钟：

```
$ mongo arete:30100
> rs.initiate()
> rs.add("arete:30101")
> rs.add("arete:30102", {arbiterOnly: true})
```

最后，在每个副本集上通过Shell运行`rs.status()`命令，验证一下两个副本集是否正常运行。如果一切顺利，就可以准备启动配置服务器了。[②]现在，创建每个配置服务器的数据目录，通过`configsvr`选项启动各个配置服务器的`mongod`进程。

```
$ mkdir /data/config-1
$ mongod --configsvr --dbpath /data/config-1 --port 27019 \
  --logpath /data/config-1.log --fork --nojournal

$ mkdir /data/config-2
$ mongod --configsvr --dbpath /data/config-2 --port 27020 \
  --logpath /data/config-2.log --fork --nojournal

$ mkdir /data/config-3
$ mongod --configsvr --dbpath /data/config-3 --port 27021 \
  --logpath /data/config-3.log --fork --nojournal
```

用Shell连接或者查看日志文件，确保每台配置服务器都已启动并已正常运行，并验证每个进程都在监听配置的端口。查看每台配置服务器的日志，应该能看到这样的内容：

```
Wed Mar  2 15:43:28 [initandlisten] waiting for connections on port 27020
Wed Mar  2 15:43:28 [websvr] web admin interface listening on port 28020
```

如果每个配置服务器都在运行了，那么就能继续下一步，启动`mongos`。必须用`configdb`选项来启动`mongos`，它接受一个用逗号分隔的配置服务器地址列表：[③]

```
$ mongos --configdb arete:27019,arete:27020,arete:27021 \
  --logpath /data/mongos.log --fork --port 40000
```

---

① `arete`是本地主机的名字。

② 同样的，如果是运行在Windows上，忽略`--fork`和`–logpath`选项，在新窗口里启动各个`mongod`。

③ 在配置列表时要小心，不要在配置服务器地址间加入空格。

2. 配置集群

现在已经准备好了所有的组件，是时候来配置集群了。先从连接mongos开始。为了简化任务，可以使用分片辅助方法，它们是全局sh对象上的方法。要查看可用辅助方法的列表，请运行sh.help()。

你将键入一系列配置命令，先从addshard命令。该命令的辅助方法是sh.addShard()，它接受一个字符串，其中包含副本集名称，随后是两个或多个要连接的种子节点地址。这里你指定了两个先前创建的副本集，用的是每个副本集中非仲裁节点的地址：

```
$ mongo arete:40000
> sh.addShard("shard-a/arete:30000,arete:30001")
  { "shardAdded" : "shard-a", "ok" : 1 }
> sh.addShard("shard-b/arete:30100,arete:30101")
  { "shardAdded" : "shard-b", "ok" : 1 }
```

如果命令执行成功，命令的响应中会包含刚添加的分片的名称。可以检查config数据库的shards集合，看看命令的执行效果。你使用了getSiblingDB()方法来切换数据库，而非use命令：

```
> db.getSiblingDB("config").shards.find()
{ "_id" : "shard-a", "host" : "shard-a/arete:30000,arete:30001" }
{ "_id" : "shard-b", "host" : "shard-b/arete:30100,arete:30101" }
```

listshards命令会返回相同的信息，这是一个快捷方式：

```
> use admin
> db.runCommand({listshards: 1})
```

在报告分片配置时，Shell的sh.status()方法能很好地总结集群的情况。现在就来试试。

下一步配置是开启一个数据库上的分片，这是对任何集合进行分片的先决条件。应用程序的数据库名为cloud-docs，可以像下面这样开启分片：

```
> sh.enableSharding("cloud-docs")
```

和以前一样，可以检查config里的数据查看刚才所做的变更。config数据库里有一个名为databases的集合，其中包含了一个数据库的列表。每个文档都标明了数据库主分片的位置，以及它是否分区（是否开启了分片）：

```
> db.getSiblingDB("config").databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }
```

现在你要做的就是分片spreadsheets集合。在对集合进行分片时，要定义一个分片键。这里将使用组合分片键{username: 1, _id: 1}，因为它能很好地分布数据，还能方便查看和理解块的范围：

```
> sh.shardCollection("cloud-docs.spreadsheets", {username: 1, _id: 1})
```

同样，可以通过检查config数据库来验证分片集合的配置：

```
> db.getSiblingDB("config").collections.findOne()
{
  "_id" : "cloud-docs.spreadsheets",
  "lastmod" : ISODate("1970-01-16T00:50:07.268Z"),
  "dropped" : false,
```

9

```
  "key" : {
    "username" : 1,
    "_id" : 1
  },
  "unique" : false
}
```

分片集合的定义可能会提醒你几点；它看起来和索引定义有几分相似之处，尤其是有那个unique键。在对一个空集合进行分片时，MongoDB会在每个分片上创建一个与分片键对应的索引。[①]可以直接连上分片，运行getIndexes()方法进行验证。此处，你可以连接到第一个分片，方法的输出包含分片键索引，正如预料的那样：

```
$ mongo arete:30000
> use cloud-docs
> db.spreadsheets.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "_id" : 1
    },
    "v" : 0
  },
  {
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "username" : 1,
      "_id" : 1
    },
    "name" : "username_1__id_1",
    "v" : 0
  }
]
```

一旦完成了集合的分片，分片集群就准备就绪了。现在可以向集群写入数据，数据将分布到各分片上。下一节里我们会了解到它是如何工作的。

9.2.2　写入分片集群

我们将向分片集合写入数据，这样你才能观察块的排列与移动。块是MongoDB分片的要素。每个示例文档都表示了一个电子表格，看起来是这样的：

```
{
  _id: ObjectId("4d6f29c0e4ef0123afdacaeb"),
  filename: "sheet-1",
  updated_at: new Date(),
  username: "banks",
  data: "RAW DATA"
}
```

① 如果是在对现有集合进行分片，必须在运行shardcollection命令前创建一个与分片键对应的索引。

请注意，data字段会包含一个5 KB的字符串以模拟原始数据。

本书的源代码中包含一个Ruby脚本，你可以用它向集群写入文档数据。该脚本接受一个循环次数作为参数，每个循环里都会为200个用户各插入5 KB的文档。脚本的源码如下：

```
require 'rubygems'
require 'mongo'
require 'names'

@con  = Mongo::Connection.new("localhost", 40000)
@col  = @con['cloud-docs']['spreadsheets']
@data = "abcde" * 1000

def write_user_docs(iterations=0, name_count=200)
  iterations.times do |n|
    name_count.times do |n|
      doc = { :filename => "sheet-#{n}",
              :updated_at => Time.now.utc,
              :username => Names::LIST[n],
              :data => @data
            }
      @col.insert(doc)
    end
  end
end

if ARGV.empty? || !(ARGV[0] =~ /^d+$/)
  puts "Usage: load.rb [iterations] [name_count]"
else
  iterations = ARGV[0].to_i

  if ARGV[1] && ARGV[1] =~ /^d+$/
    name_count = ARGV[1].to_i
  else
    name_count = 200
  end

  write_user_docs(iterations, name_count)
end
```

如果手头有脚本，可以在命令行里不带参数运行脚本，它会循环一次，插入200个值：

```
$ ruby load.rb
```

现在，通过Shell连接mongos。如果查询spreadsheets集合，你会发现其中包含200个文档，总大小在1 MB左右。还可以查询一个文档，但要排除data字段（你不想在屏幕上输出5 KB文本吧）。

```
$ mongo arete:40000
> use cloud-docs
> db.spreadsheets.count()
200
> db.spreadsheets.stats().size
1019496
> db.spreadsheets.findOne({}, {data: 0})
{
  "_id" : ObjectId("4d6d6b191d41c8547d0024c2"),
```

```
  "username" : "Cerny",
  "updated_at" : ISODate("2011-03-01T21:54:33.813Z"),
  "filename" : "sheet-0"
}
```

现在，可以检查一下整个分片范围里发生了什么，切换到config数据库，看看块的个数：

```
> use config
> db.chunks.count()
1
```

目前只有一个块，让我们看看它什么样：

```
> db.chunks.findOne()
{
  "_id" : "cloud-docs.spreadsheets-username_MinKey_id_MinKey",
  "lastmod" : {
    "t" : 1000,
    "i" : 0
  },
  "ns" : "cloud-docs.spreadsheets",
  "min" : {
    "username" : { $minKey : 1 },
    "_id" : { $minKey : 1 }
  },
  "max" : {
    "username" : { $maxKey : 1 },
    "_id" : { $maxKey : 1 }
  },
  "shard" : "shard-a"
}
```

你能说出这个块所表示的范围吗？如果只有一个块，那么它的范围是这个分片集合。这是由min和max字段标识的，这些字段通过$minKey和$maxKey限定了块的范围。

MINKEY与MAXKEY

作为BSON类型的边界，$minKey与$maxKey常用于比较操作之中。$minKey总是小于所有BSON类型，而$maxKey总是大于所有BSON类型。因为给定的字段值能包含各种BSON类型，所以在分片集合的两端，MongoDB使用这两个类型来标记块的端点。

通过向spreadsheets集合添加更多数据，你能看到更有趣的块范围。还是使用之前的Ruby脚本，但这次要循环100次，向集合中插入20 000个文档，总计100 MB：

```
$ ruby load.rb 100
```

可以像下面这样验证插入是否成功：

```
> db.spreadsheets.count()
20200
> db.spreadsheets.stats().size
103171828
```

样本插入速度

注意，向分片集群插入数据需要花好几分钟时间。速度如此之慢有三个原因。首先，每次插入都要与服务器交互一次，而在生产环境中可以使用批量插入。其次，你是在使用Ruby进行插入，Ruby的BSON序列器要比其他某些驱动的慢。最后，也是最重要的，你是在一台机器上运行所有分片节点的，这为磁盘带来了巨大的负担，因为四个节点正在同时向磁盘写入数据（两个副本集的主节点，以及两个副本集的从节点）。有理由相信，在适当的生产环境部署中，插入的速度会快许多。

插入了这么多数据之后，现在肯定有不止一个块了。可以统计chunks集合的文档数快速检查块的状态：

```
> use config
> db.chunks.count()
10
```

运行sh.status()能看到更详细的信息，该方法会输出所有的块以及它们的范围。简单起见，我只列出头两个块的信息：

```
> sh.status()
sharding version: { "_id" : 1, "version" : 3 }
 shards:
 { "_id": "shard-a", "host": "shard-a/arete:30000,arete:30001" }
 { "_id": "shard-b", "host": "shard-b/arete:30100,arete:30101" }

 databases:
 { "_id": "admin", "partitioned": false, "primary": "config" }
 { "_id": "test",  "partitioned": false, "primary": "shard-a" }
 { "_id": "cloud-docs", "partitioned": true,  "primary": "shard-b" }
    shard-a 5
    shard-b 5
  { "username": { $minKey : 1 }, "_id" : { $minKey : 1 } } --
     >>   { "username": "Abdul",
    "_id": ObjectId("4e89ffe7238d3be9f0000012") }
  on: shard-a { "t" : 2000, "i" : 0 }

   { "username" : "Abdul",
     "_id" : ObjectId("4e89ffe7238d3be9f0000012") } -->> {
     "username" : "Buettner",
     "_id" : ObjectId("4e8a00a0238d3be9f0002e98") }
   on : shard-a { "t" : 3000, "i" : 0 }
```

情况明显不同了，现在有10个块了。当然，每个块所表示的是一段连续范围的数据。可以看到第一个块由$minKey到Abdul的文档构成，第二个块由Abdul到Buettner的文档构成。[①]不仅块变多了，块还迁移到了第二个分片上。通过sh.status()的输出能看到这个变化，但还有更简单的方法：

① 如果你是跟着示例一路做下来的，会发现自己的块分布与示例稍有不同。

```
> db.chunks.count({"shard": "shard-a"})
5
> db.chunks.count({"shard": "shard-b"})
5
```

集群的数据量还不大。拆分算法表明会经常发生数据拆分，正如你所见，这能在早期就实现数据和块的均匀分布。从现在开始，只要写操作在现有块范围内保持均匀分布，就不太会发生迁移。

早期块拆分

分片集群会在早期积极进行块拆分，以便加快数据在分片中的迁移。具体说来，当块的数量小于10时，会按最大块尺寸（16 MB）的四分之一进行拆分；当块的数量在10到20之间时，会按最大块尺寸的一半（32 MB）进行拆分。

这种做法有两个好处。首先，这会预先创建很多块，触发一次迁移。其次，这次迁移几乎是无痛的，因为块的尺寸越小，其迁移的数据就越少。

现在，拆分的阈值会增大。通过大量插入数据，你会看到拆分是怎么缓缓减慢的，以及块是怎么增长到最大尺寸的。试着再向集群里插入800 MB数据：

```
$ ruby load.rb 800
```

这条命令会执行很长时间，因此你可能会想在启动加载进程后暂时离开吃些点心。执行完毕之后，总数据量比以前增加了8倍。但是如果查看分块状态，你会发现块的数量差不多只是原来的两倍：

```
> use config
> db.chunks.count()
21
```

由于块的数量变多了，块的平均范围就变小了，但是每个块都会包含更多数据。举例来看，集合里的第一个块的范围只是`Abbott`到`Bender`，但它的大小已经接近60 MB了。因为目前块的最大尺寸是64 MB，所以如果继续插入数据，很快就能看到块的拆分了。

另一件值得注意的事情是块的分布还是很均匀的，就和之前一样：

```
> db.chunks.count({"shard": "shard-a"})
11
> db.chunks.count({"shard": "shard-b"})
10
```

尽管刚才在插入800 MB数据时块的数量增加了，但你还是可以猜到没有发生迁移；一个可能的情况是每个原始块被一拆为二，期间还有一次额外的拆分。可以查询`config`数据库的`changelog`集合加以验证：

```
> db.changelog.count({what: "split"})
20
> db.changelog.find({what: "moveChunk.commit"}).count()
6
```

这符合我们的猜测。一共发生了20次拆分，产生了20个块，但只发生了6次迁移。要再深入了解一下究竟发生了什么，可以查看变更记录的具体条目。举例来说，以下条目记录了第一次的块移动：

```
> db.changelog.findOne({what: "moveChunk.commit"})
{
  "_id" : "arete-2011-09-01T20:40:59-2",
  "server" : "arete",
  "clientAddr" : "127.0.0.1:55749",
  "time" : ISODate("2011-03-01T20:40:59.035Z"),
  "what" : "moveChunk.commit",
  "ns" : "cloud-docs.spreadsheets",
  "details" : {
    "min" : {
      "username" : { $minKey : 1 },
      "_id" : { $minKey : 1 }
    },
    "max" : {
      "username" : "Abbott",
      "_id" : ObjectId("4d6d57f61d41c851ee000092")
    },
    "from" : "shard-a",
    "to" : "shard-b"
  }
}
```

这里可以看到块从shard-a移到了shard-b。总的来说，在变更记录里找到的文档可读性都比较好。在深入了解分片并打算打造自己的分片集群之时，配置变更记录是了解拆分和迁移行为的优秀材料，应该经常看看它。

9.3 分片集群的查询与索引

从应用程序的角度来看，查询分片集群和查询单个mongod没什么区别。这两种情况下，查询接口和迭代结果集的过程是一样的。但在外表之下，两者还是有区别的，有必要了解一下其中的细节。

9.3.1 分片查询类型

假设正在查询一个分片集群，为了返回一个恰当的查询响应，mongos要与多少个分片进行交互？稍微思考一下，就能发现这与分片键是否出现在查询选择器里有关。还记得吗？配置服务器（就是mongos）维护了一份分片范围的映射关系，就是我们在本章早些时候看到的块。如果查询包含分片键，那么mongos通过块数据能很快定位哪个分片包含查询的结果集。这称为针对性查询（targeted query）。

但是，如果分片键不是查询的一部分，那么查询计划器就不得不访问所有分片来完成查询。这称为全局查询或分散/聚集查询（scatter/gather query）。图9-3对这两种查询做了描述。

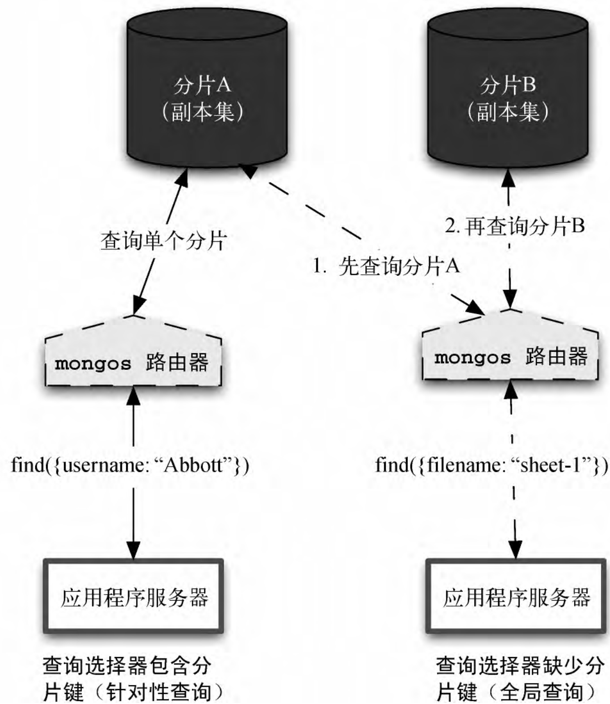


图9-3　针对副本集的针对性查询与全局查询

针对任意指定的分片集群查询，explain命令能显示其详细查询路径。让我们先来看一个针对性查询，此处要查询位于集合第一个块里的文档。

```
> selector = {username: "Abbott",
  "_id" : ObjectId("4e8a1372238d3bece8000012")}
> db.spreadsheets.find(selector).explain()
{
  "shards" : {
    "shard-b/arete:30100,arete:30101" : [
      {
        "cursor" : "BtreeCursor username_1__id_1",
        "nscanned" : 1,
        "n" : 1,
        "millis" : 0,
        "indexBounds" : {
          "username" : [
            [
              "Abbott",
              "Abbott"
            ]
          ],
          "_id" : [
            [
              ObjectId("4d6d57f61d41c851ee000092"),
```

```
                ObjectId("4d6d57f61d41c851ee000092")
              ]
            ]
          }
        }
      ]
    },
    "n" : 1,
    "nscanned" : 1,
    "millisTotal" : 0,
    "numQueries" : 1,
    "numShards" : 1
  }
```

explain的结果清晰地说明查询命中了一个分片——分片B，返回了一个文档。[1]查询计划器很聪明地使用了分片键前缀的子集来路由查询。也就是说你也可以单独根据用户名进行查询：

```
> db.spreadsheets.find({username: "Abbott"}).explain()
{
  "shards" : {
    "shard-b/arete:30100,arete:30101" : [
      {
        "cursor" : "BtreeCursor username_1__id_1",
        "nscanned" : 801,
        "n" : 801,
      }
    ]
  },
  "n" : 801,
  "nscanned" : 801,
  "numShards" : 1
}
```

该查询总共返回了801个用户文档，但仍然只访问了一个分片。

那么全局查询又会怎么样呢？也可以方便地使用explain命令。下面就是一个根据filename字段进行查询的例子，其中既没有用到索引，也没有用到分片键：

```
> db.spreadsheets.find({filename: "sheet-1"}).explain()
{
  "shards" : {
    "shard-a/arete:30000,arete:30002,arete:30001" : [
      {
        "cursor" : "BasicCursor",
        "nscanned" : 102446,
        "n" : 117,
        "millis" : 85,
      }
    ],
    "shard-b/arete:30100,arete:30101" : [
      {
```

① 注意，简单起见，在这个执行计划以及接下来的执行计划里，我省略了很多字段。

```
        "cursor" : "BasicCursor",
        "nscanned" : 77754,
        "nscannedObjects" : 77754,
        "millis" : 65,
      }
    ]
  },
  "n" : 2900,
  "nscanned" : 180200,
  "millisTotal" : 150,
  "numQueries" : 2,
  "numShards" : 2
}
```

如你所想，该全局查询在两个分片上都进行了表扫描。如果该查询与你的应用程序有关，你一定想在filename字段上增加一个索引。无论哪种情况，它都会搜索整个集群以返回完整结果。

一些查询要求并行获取整个结果集。例如，假设想根据修改时间对电子表格进行排序。这要求在mongos路由进程里合并结果。没有索引，这样的查询会非常低效，并且会屡遭禁止。因此，在下面这个查询最近创建文档的例子里，你会先创建必要的索引：

```
> db.spreadsheets.ensureIndex({updated_at: 1})
> db.spreadsheets.find({}).sort({updated_at: 1}).explain()
{
  "shards" : {
    "shard-a/arete:30000,arete:30002" : [
      {
        "cursor" : "BtreeCursor updated_at_1",
        "nscanned" : 102446,
        "n" : 102446,
        "millis" : 191,
      }
    ],
    "shard-b/arete:30100,arete:30101" : [
      {
        "cursor" : "BtreeCursor updated_at_1",
        "nscanned" : 77754,
        "n" : 77754,
        "millis" : 130,
      }
    ]
  },
  "n" : 180200,
  "nscanned" : 180200,
  "millisTotal" : 321,
  "numQueries" : 2,
  "numShards" : 2
}
```

正如预期的那样，游标扫描了每个分片的updated_at索引，以此返回最近更新的文档。

更有可能出现的查询是返回某个用户最新修改的文档。同样，你要创建必要的索引，随后发起查询：

```
> db.spreadsheets.ensureIndex({username: 1, updated_at: -1})
> db.spreadsheets.find({username: "Wallace"}).sort(
  {updated_at: -1}).explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "shard-1-test-rs/arete:30100,arete:30101" : [
      {
        "cursor" : "BtreeCursor username_1_updated_at_-1",
        "nscanned" : 801,
        "n" : 801,
        "millis" : 1,
      }
    ]
  },
  "n" : 801,
  "nscanned" : 801,
  "numQueries" : 1,
  "numShards" : 1
}
```

关于这个执行计划，有几个需要注意的地方。首先，该查询指向了单个分片。因为你指定了分片键，所以查询路由器可以找出哪个分片包含了相关的块。随后你就会发现排序并不需要访问所有的分片；当排序查询中包含分片键，所要查询的分片数量通常都能有所减少。本例中，只需访问一个分片，也能想象类似的查询，即需要访问几个分片，所访问的分片数量少于分片总数。

第二个需要注意的地方是分片使用了{username: 1,updated_at: -1}索引来执行查询。这说明了一个很重要的内容,即分片集群是如何处理查询的。通过分片键将查询路由给指定分片，一旦到了某个分片上，由分片自行决定使用哪个索引来执行该查询。在为应用程序设计查询和索引时，请牢记这一点。

9.3.2 索引

你刚看了一些例子，其中演示了索引查询是如何在分片集群里工作的。有时，如果不确定某个查询是怎么解析的，可以试试explain()。通常这都很简单，但是在运行分片集群时，有几点关于索引的内容应该牢记于心，下面我会逐个进行说明。

(1) 每个分片都维护了自己的索引。这点应该是显而易见的，当你在分片集合上声明索引时，每个分片都会为它那部分集合构建独立的索引。例如，在上一节里，你通过mongos发起了db.spreasheets.ensureIndex()命令，每一个分片都单独处理了索引创建命令。

(2) 由此可以得出一个结论，每个分片上的分片集合都应该拥有相同的索引。如果不是这样的话，查询性能会很不稳定。

(3) 分片集合只允许在_id字段和分片键上添加唯一性索引。其他地方不行，因为这需要在分片间进行通信，实施起来很复杂，而且相信这么做速度也很慢，没有实现的价值。

一旦理解了如何进行查询的路由选择，以及索引是如何工作的，你应该就能针对分片集群写

出漂亮的查询和索引了。第7章里几乎所有关于索引和查询优化的建议都能用得上，此外，在必要的时候，你还可以使用强大的explain()工具。

9.4　选择分片键

上面说的这些都依赖于正确选择分片键。分片键选的不好，应用程序就无法利用分片集群所提供的诸多优势。在这种情况下，插入和查询的性能都会显著下降。下决定时一定要严肃，一旦选定了分片键，就必须坚持选择，分片键是不可修改的。[①]

要让分片能提供好的体验，部分源自了解怎样才算一个好的分片键。因为这并不是很直观，所以我会先描述一些不太好的分片键。这能很自然地引出对好分片键的讨论。

9.4.1　低效的分片键

一些分片键的分布性很差，而另一些则导致无法充分利用局部性原理，还有一些可能会妨碍块的拆分。本节我们会看到一些产生这种不理想状态的分片键。

1. 分布性差

BSON对象ID是每个MongoDB文档的默认主键。乍一看，一个与MongoDB核心如此接近的数据类型很有可能成为候选的分片键。然而，我们不能被表像蒙蔽。回想一下，所有对象ID中最重要的组成部分是时间戳，也就是说对象ID始终是升序的。遗憾的是，升序的值对分片键而言是很糟糕的。

要了解升序分片键的问题，你要牢记分片是基于范围的。使用升序的分片键之后，所有最近插入的文档都会落到某个很小的连续范围内。用分片的术语来说，就是这些插入都会被路由到一个块里，也就是被路由到单个分片上。这实际上抵消了分片一个很大的好处：将插入的负载自动分布到不同机器上。[②]结论已经很清楚了，如果想让插入负载分不到多个分片上，就不能使用升序分片键，你需要某些随机性更强的东西。

2. 缺乏局部性

升序分片键有明确的方向，完全随机的分片键则根本没有方向。前者无法分散插入，而后者则可能是将插入分得太散。这点可能会违背你的直觉，因为分片的目的就是要分散读写操作。我们可以通过一个简单的思想实验对此做出说明。

假设分片集合里的每个文档都包含一个MD5，而且MD5字段就是分片键。因为MD5的值会随着文档的不同随机变化，所以该分片键能确保插入的文档均匀分布在集群的所有分片上，这样很好。但是再仔细一想，对每个分片的MD5字段索引进行的插入又会怎么样？因为MD5是完全随机的，在每次插入过程中，索引中的每个虚拟内存分页都有可能（同等可能性）被访问到。实

① 注意，一旦创建了分片键，没有什么好办法来修改它。你最好用合适的键再创建一个新的分片集合，从老分片集合里把数据导出来，再把它们还原到新集合里。

② 注意，升序的分片键不会影响到更新，只要文档都是随机更新的。

际上，这就意味着索引必须总是能装在内存里，如果索引和数据不断增多，超出了物理内存的限制，那些会降低性能的页错误是不可避免的。

这基本就是一个*局部引用性*（locality of reference）问题。局部的概念，至少在这里是指任意给定时间间隔内所访问的数据基本都是有关系的；这能用来进行相关优化。例如，虽然对象ID是个糟糕的分片键，但它们提供了很好的局部性，因为它们是升序的。也就是说，对索引的连续插入都会发生在最近使用的虚拟内存分页里；因此，在任意时刻内存里只要有一小部分索引就可以了。

举个不太抽象的例子，想象一下，假设你的应用程序允许用户上传照片，每张照片的元数据都保存在某个分片集合的一个文档里。现在，假设用户批量上传了100张照片。如果分片键是完全随机的，那么数据库就无法利用局部性；对索引的插入会发生在100个随机的地方。但是，如果我们假设分片键是用户的ID，又会怎么样？此时，每次写索引基本都会发生在同一个地方，因为插入的每个文档都拥有相同的用户ID值。这就利用到了局部性，你也能体会到潜在的显著性能提升。

随机分片键还有另一个问题：对这个键的任意一个有意义的范围查询都会被发送到所有分片上。还是刚才那个分片照片集合，如果你想让应用显示某个用户最近创建的10张照片（这是一个很普通的查询），随机分片键仍会要求把该查询发到所有的分片上。正如你将在下文里看到的那样，较粗粒度的分片键能让这样的范围查询落到单个分片上。

3. 无法拆分的块

如果随机分片键和升序分片键都不好用，那么下一个显而易见的选择就是粗粒度分片键，用户ID就是很好的例子。如果根据用户ID对照片集合进行分片，你可以预料到插入会分布在各个分片上，因为无法预知哪个用户何时会插入数据。这样一来，粗粒度分片键也能拥有随机性，还能发挥分片集群的优势。

粗粒度分片键的第二个好处是能通过局部引用性带来效率的提升。当某个用户插入100个照片元数据文档，基于用户ID字段的分片键能确保这些插入都落到同一个分片上，并几乎能写入索引的同一部分。这样的效率很高。

粗粒度分片键在分布性和局部性方面表现的都很好，但它也有一个很难解决的问题：块有可能无限制地增长。这怎么可能？想想基于用户ID的示例分片键，它能提供的最小块范围是什么？是用户ID，不可能再小了。每个数据集都有可能存在异常情况，这时就会有问题。假设有几个特殊用户，他们保存的照片数量超过普通用户数百万。系统能将一个用户的照片拆分到多个块里么？答案是不能！这个块不能拆分。这对分片集群是个危害，因为这会造成分片间数据不均衡的情况。

显然，理想的分片键应该结合了粗粒度分片键与细粒度分片键两者的优势。下一节里你就能一睹它的芳容。

9.4.2 理想的分片键

通读上一节，你应该已经清楚地知道理想的分片键应该能够：

(1) 将插入数据均匀分布到各个分片上；

(2) 保证CRUD操作能够利用局部性；

(3) 有足够的粒度进行块拆分。

满足这些要求的分片键通常由两个字段组成，第一个是粗粒度的，第二个粒度较细。电子表格示例的分片键就是一个不错的例子，你声明了一个复合分片键{username: 1, _id: 1}。当不同的用户向集群插入数据时，可以预计到大多数（并非全部）情况下，一个用户的电子表格会在单个分片上。就算某个用户的文档落在多个分片上，分片键里那个唯一的_id字段也能保证对任意一个文档的查询和更新始终能指向单个分片。如果需要对某个用户的数据执行更复杂的查询，可以保证查询只会被路由到包含该用户数据的那些分片上。

最重要的是分片键{username: 1, _id: 1}保证了块始终是能继续拆分的，哪怕用户创建了大量文档，情况也是如此。

再举个例子，假设正在构建一个网站分析系统。正如将在附录B里看到的那样，针对此类系统，一个不错的数据模型是每个网页每月保存一个文档。随后，在那个文档内保存该月每天的数据，每次访问某个页面就增加一些计数器字段的值等。下面是与分片键选择有关的示例分析文档字段：

```
{ _id: ObjectId("4d750a90c35169d10fc8c982"),
  domain: "org.mongodb",
  url: "/downloads",
  period: "2011-12"
}
```

针对包含此类文档的分片集群，最简单的分片键包含每个网页的域名，随后是URL：{domain: 1, url: 1}。所有来自指定域的页面通常都能落在一个分片上，但是一些特殊的域拥有大量页面，在必要时仍会被拆分到多个分片上。

9.5　生产环境中的分片

在生产环境里部署分片集群时，面前会出现很多选择和挑战。这里我会描述几个推荐的部署拓扑，针对常见的部署问题做出解答。我们随后还会考虑一些服务器管理方面的问题，包括监控、备份、故障转移和恢复。

9.5.1　部署与配置

一开始很难搞定分片集群的部署与配置，下文是一份指南，介绍了轻松组织并配置集群的方法。

1. 部署拓扑

要运行示例MongoDB分片集群，你一共要启动九个进程（每个副本集三个mongod，外加三个配置服务器）。乍一看，这个数字有点吓人。一开始用户会假设在生产环境里运行两个分片的集群要有九台独立的机器。幸运的是，实际需要的机器要少很多，看一下集群中各组件所要求的资源就能知道为什么了。

首先考虑一下副本集，每个成员都包含分片的完整数据副本，可能是主节点，也可能是从节点。这些进程总是要求有足够的磁盘空间来保存数据，要有足够的内存高效地提供服务。因此，复制`mongod`是分片集群中最资源密集型的进程，必须占用独立的机器。

那副本集的仲裁节点呢？这些进程只保存副本集的配置数据，这些数据就放在一个文档里。所以，仲裁节点开销很少，当然也就不需要自己的服务器了。

接下来是配置服务器，它们同样只保存相对较少的数据。举例来说，配置服务器上管理示例副本集的数据一共也就大约30 KB。如果假设这些数据会随着分片集群数据的增长而线性增长，那么1 TB的分片集群可能仅会对应30 MB数据。[①]也就是说配置服务器同样不需要有自己的机器。但是，考虑到配置服务器所扮演的重要角色，一些用户更倾向于为它们提供一些机器（或虚拟机）。

根据你对副本集和分片集群的了解，可以列出部署分片集群的最低要求。

(1) 副本集的每个成员，无论是完整的副本节点还是仲裁节点，都需要放在不同的机器上。

(2) 每个用于复制的副本集成员都需要有自己的机器。

(3) 副本集的仲裁节点是很轻量级的，和其他进程共用一台机器就可以了。

(4) 配置服务器也可以选择与其他进程共用一台机器。唯一的硬性要求是配置集群中的所有配置服务器都必须放在不同的机器上。

你可能感觉要满足这些规则会引起逻辑问题。我们将运用这些规则：针对示例的两分片集群，你会看到两个合理的部署拓扑。第一个拓扑只需要四台机器，图9-4里描绘了进程的分布情况。

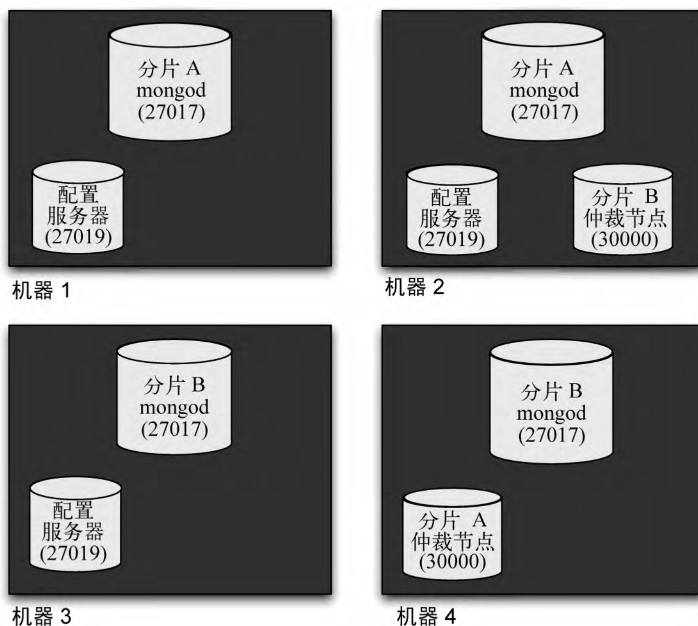


图9-4　部署在四台机器上的两分片集群

① 这是一个相当保守的估计，真实值可能会小得多。

这个配置满足了刚才所说的所有规则。在每台机器上占主导地位的是各分片的复制节点。剩下的进程经过了精心安排，所有的三个配置服务器和每个副本集的仲裁节点都部署在了不同的机器上。说起容错性，该拓扑能容忍任何一台机器发生故障。无论哪台机器发生了故障，集群都能继续处理读写请求。如果发生故障的机器正好运行了一个配置服务器，那么所有的块拆分和迁移都会暂停。[①]幸运的是，暂停分片操作基本不会影响分片集群的工作；在损失的机器恢复后，就能进行拆分和迁移了。

这是两分片集群的*最小推荐配置*。但是，那些要求最高可用性和最快恢复途径的应用程序需要一些更强健的东西。正如上一章里讨论的那样，包含两个副本和一个仲裁节点的副本集在恢复时是很脆弱的。如果有三个节点，就能降低恢复时的脆弱程度，还能让你在从数据中心里部署一个节点，用于灾难恢复。图9-5是一个强壮的两分片集群拓扑。每个分片都包含三节点的副本集，每个节点都包含数据的完整副本。为了进行灾难恢复，从每个分片里抽一个节点，加上一个配置服务器，部署在从数据中心；要保证那些节点不会变成主节点，可以将它们的优先级设置为0。

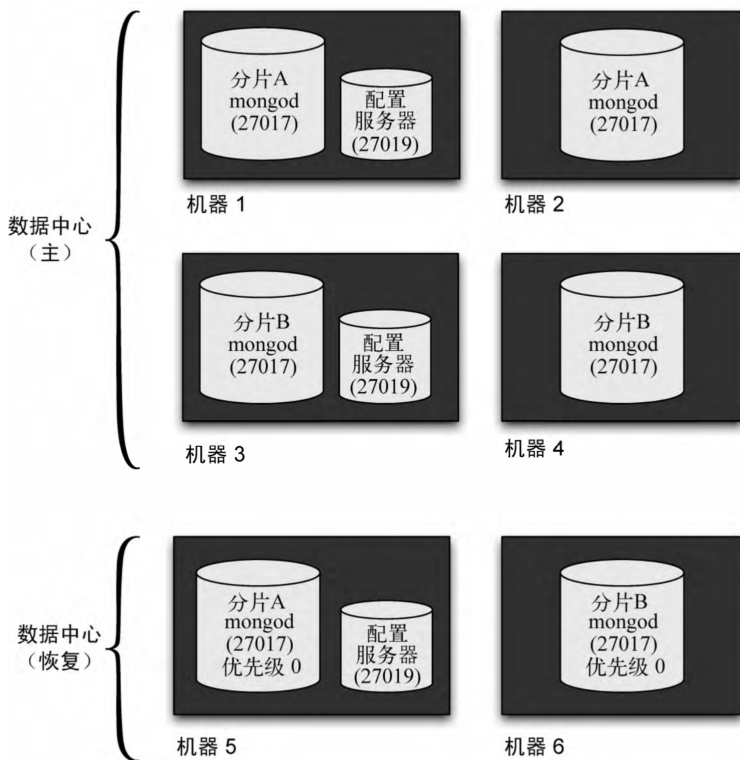


图9-5　部署在两个数据中心、六台机器上的两分片集群

①在发生任何分片操作时，所有的三台配置服务器都必须在线。

用了这个配置，每个分片都会被复制两次，而非仅一次。此外，当主数据中心发生故障时，从数据中心拥有重建分片集群所需的全部数据。

数据中心故障

最有可能发生的数据中心故障是电力中断。在没有开启Journaling日志的情况下运行MongoDB服务器时，电力中断就意味着非正常关闭MongoDB服务器，可能会损坏数据文件。发生这种故障时，唯一可靠的恢复途径是数据库修复，一个保证停机时间的漫长过程。

大多数用户只将整个集群部署在一个数据中心里，这对大量应用程序来说都没问题。这种情况下的主要预防措施是，至少在每个分片的一个节点以及一台配置服务器上开启Journaling日志。在电力供应恢复时，这能极大地提高恢复速度。第10章里会涉及Journaling日志的相关内容。

尽管如此，一些故障更加严重。电力中断有时能持续几天。洪水、地震，以及其他自然灾害能完全摧毁数据中心。对于那些想在此类故障中进行快速恢复的用户而言，他们必须跨多个数据中心部署分片集群。

哪种分片拓扑最适合你的应用程序，这种决策总是基于一系列与你能容忍的停机时间有关的考虑，比如根据MTR（Mean Time to Recovery，平均恢复时间）进行评估。考虑潜在的故障场景，并模拟它们。如果一个数据中心发生故障，考虑一下它对应用程序（或业务）的影响。

2. 配置注意事项

下面是一些与配置分片集群相关的注意事项。

- 估计集群大小

用户经常想知道要部署多少个分片，每个分片应该有多大。当然，这个问题的答案取决于所在的环境。如果是部署在亚马逊的EC2上，在超过最大的可用实例前都不应该进行分片。在本书编写时，最大的EC2节点有68 GB内存。如果运行在自己的硬件上，你还可以拥有更大的机器。在数据量达到100 GB之前都不进行分片，这是很合理的。

当然，每增加一个分片都会引入额外的复杂性，每个分片都要求进行复制。所以说，少数大分片比大量小分片要好。

- 对现有集合进行分片

你可以对现有集合进行分片，如果花了很多时间才将数据分布到各分片里，请不要大惊小怪的。每次只能做一轮均衡，迁移过程中每分钟只能移动大约100~200 MB数据。因此，对一个50 GB的集合进行分片大约需要八个小时，其中还可能牵涉一定的磁盘活动。此外，在对这样的大集合进行初始分片时，可能还要手动拆分以加速分片过程，因为拆分是由插入触发的。

说到这里，应该已经很清楚了，在最后时刻对一个集合进行分片并不是处理性能问题的好办法。如果你计划在未来某个时刻对集合进行分片，考虑到可以预见的性能下降，应该提前进行分片。

- 在初始加载时预拆分块

如果你有一个很大的数据集需要加载到分片集合里，并且知道数据分布的规律，那么可以通

过对块的预拆分和预迁移节省很多时间。举个例子，假设你想要把电子表格导入到一个新的MongoDB分片集群里。可以在导入时先拆分块，随后将它们迁移到分片里，借此保证数据是均匀分布的。你能用split和moveChunk命令实现这个目标，它们的辅助方法分别是sh.splitAt()和sh.moveChunks()。

下面是一个手动块拆分的例子。你发出split命令，指定你想要的集合，随后指明拆分点：

```
> sh.splitAt( "cloud-docs.spreadsheets",
  { "username" : "Chen", "_id" : ObjectId("4d6d59db1d41c8536f001453") })
```

命令运行时会定位到某个块，而这个块逻辑上包含username是Chen并且_id是ObjectId("4d6d59db1d41c8536f001453")的文档[①]。该命令随后会根据这个点来拆分块，最后得到两个块。你能像这样继续拆分，直到拥有数据良好分布的块集合。你还要确保创建足够数量的块，让平均块大小保持在64 MB的拆分阈值以内。所以，如果想加载1GB数据，应该计划创建大约20个块。

第二步是确定所有分片都拥有数量相当的块。因为所有的块最初都在一个分片上，你需要移动它们。可以使用moveChunk命令来移动块。辅助方法能简化这个过程：

```
> sh.moveChunk("cloud-docs.spreadsheets", {username: "Chen"}, "shardB")
```

这句语句的意思是把逻辑上包含文档{username: "Chen"}的块移动到分片B上。

9.5.2　管理

我将简单介绍一些分片管理的知识，让本章内容更充实一些。

1. 监控

分片集群是整个体系中比较复杂的一块，正因此，你应该严密监控它。在任何mongos上都可以运行serverStatus和currentOp()命令，命令的输出能反映所有分片的聚合统计信息。在下一章里我将更具体地讨论这些命令。

除了聚合服务器的统计信息，你还希望能监控块的分布和各个块的大小。正如在示例集群中看到的那样，所有的信息都保存在config数据库里。如果发现不平衡的块或者未经确认的块增长，可以通过split和movechunk命令处理这些情况。或者，也可以查看日志，检查均衡操作是否出于某些原因被停止了。

2. 手动分区

有一些情况下，你可能希望手动对线上分片集群的块进行拆分和迁移。例如，自MongoDB v2.0起，均衡器并不会直接考虑某个分片的负载。很明显，一个分片的写越多，它的块就越大，最终就会造成迁移。但是，不难想象你可以通过迁移块来减轻分片的负载。movechunk命令在这种情况下同样很有帮助。

3. 增加一个分片

如果你决定要增加更多容量，可以使用与先前一样的方法向现有集群添加新的分片：

---

① 注意，并不需要存在这样一个文档。事实上，你正在对一个空集合做拆分。

```
sh.addShard("shard-c/rs1.example.net:27017,rs2.example.net:27017")
```

使用这种方式增加容量时，要注意向新分片迁移数据所花费的时间。如前所述，预计的迁移速度是每分钟100~200 MB。这意味着如果需要向分片集群增加容量，你应该早在性能下降以前就开始行动。要决定何时需要添加新分片，考虑一下数据集的增长速率。很明显，你希望将索引和工作集保持在内存里。因此，最好在索引和工作集达到现有分片内存90%之前的几个星期就开始计划添加新分片。

如果你不愿意采用此处描述的安全途径，那么就会将自己置身于痛苦之中。一旦内存里容纳不下索引和工作集，应用程序就会中止运行，尤其是那些要求很高读写吞吐量的应用程序。问题在于数据库需要在磁盘和内存之间置换分页，这会降低读写速度，后台日志操作无法放入读写队列。从这点来看，增加容量是件困难的事，因为分片之间的块迁移会增加现有分片的读负载。很明显，在数据库已经超载之时，你最后想做的还是增加负载。

说了这么多，只是为了强调你应该监控集群，在真正有需要之前就增加容量。

4. 删除分片

在一些很少见的情况下，你可能会想删除一个分片。可以通过removeshard命令进行删除：

```
> use admin
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{
  "msg" : "draining started successfully",
  "state" : "started",
  "shard" : "shard-1-test-rs",
  "ok" : 1 }
```

命令的响应说明正在从分片中移除块，它们将被重新分配到其他分片上。可以再次运行该命令来检查删除过程的状态：

```
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{
  "msg" : "draining ongoing",
  "state" : "ongoing",
  "remaining" : {
    "chunks" : 376,
    "dbs" : 3
  },
  "ok" : 1 }
```

一旦分片被清空，你还要确认将要删除的分片不是数据库的主分片。可以通过查询config.databases集合的分片成员进行检查：

```
> use config
> db.databases.find()
 { "_id" : "admin", "partitioned" : false, "primary" : "config" }
 { "_id" : "cloud-docs", "partitioned" : true, "primary" : "shardA" }
 { "_id" : "test", "partitioned" : false, "primary" : "shardB" }
```

从中可以看到，cloud-docs数据库属于shardA，而test数据库则属于shardB。因为正在删除shardB，所以需要改变test数据库的主节点。为此，可以使用moveprimary命令：

```
> db.runCommand({moveprimary: "test", to: "shard-0-test-rs" });
```

对于每个主节点是将要删除的分片的数据库，运行该命令。随后，再次对每个已清空的分片运行removeshard命令：

```
> db.runCommand({removeshard: "shard-1/arete:30100,arete:30101"})
{ "msg": "remove shard completed successfully",
  "stage": "completed",
  "host": "arete:30100",
  "ok" : 1
}
```

一旦看到删除完成，就可以安全地将已删除的分片下线了。

5. 集合去分片

虽然可以删除一个分片，但是没有正式的途径去掉集合的分片。如果真的需要这么做，最好的选择是导出集合，再用一个不同的名字将数据恢复到一个新的集合里。[①]然后就能把已经导出数据的分片集合删掉了。例如，假设foo是一个分片集合，你必须用mongodump连接mongos来导出foo集合的数据：

```
$ mongodump -h arete --port 40000 -d cloud-docs -c foo
connected to: arete:40000
DATABASE: cloud-docs   to   dump/cloud-docs
  cloud-docs.foo to dump/cloud-docs/foo.bson
     100 objects
```

该命令能把该集合导出到一个名为foo.bson的文件里，随后再用mongorestore来恢复该文件：

```
$ mongorestore -h arete --port 40000 -d cloud-docs -c bar
Tue Mar 22 12:06:12 dump/cloud-docs/foo.bson
Tue Mar 22 12:06:12    going into namespace [cloud-docs.bar]
Tue Mar 22 12:06:12    100 objects found
```

将数据移动到未分片集合之后，就可以随意删除旧的分片集合foo了。

6. 备份分片集群

要备份分片集群，你需要配置数据以及每个分片数据的副本。有两种途径来获得这些数据。第一种是使用mongodump工具，从一个配置服务器导出数据，随后再从每个单独的分片里导出数据。此外，也可以通过mongos路由器运行mongodump，一次性导出整个分片集合的数据，包括配置数据库。这种策略的主要问题是分片集合的总数据可能太大了，以至于无法导出到一台机器上。

另一种常用的备份分片集群的方法是从每个分片的一个成员里复制数据文件，再从一台配置服务器中复制数据文件。下一章里会介绍这种备份独立mongod进程和副本集的方法。你只要在每个分片和一台配置服务器上执行这个过程就可以了。

无论选择哪种备份方式，都需要确认在备份系统时没有块处在移动过程中。也就是说要停止均衡器进程。

● 停止均衡器

到目前为止，禁用均衡器就是upsert一个文档到config数据库的settings集合：

① 下一章将涉及用来进行导出和恢复的工具——mongodump和mongorestore。

```
> use config
> db.settings.update({_id: "balancer"}, {$set: {stopped: true}}, true);
```

这里一定要小心：更新了配置之后，均衡器可能仍在工作。在备份集群之前，你还需要再次确认均衡器完成了最后一轮均衡。最好的方法就是检查locks集合，找到_id是balancer的条目，确认它的状态是0。下面是一个例子：

```
> use config
> db.locks.find({_id: "balancer"})
{ "_id" : "balancer", "process" : "arete:40000:1299516887:1804289383",
  "state" : 1,
  "ts" : ObjectId("4d890d30bd9f205b29eda79e"),
  "when" : ISODate("2011-03-22T20:57:20.249Z"),
  "who" : "arete:40000:1299516887:1804289383:Balancer:846930886",
  "why" : "doing balance round"
}
```

任何大于0的状态值都说明均衡仍在进行中。process字段显示了负责组织协调均衡的mongos所运行在的计算机的主机名和端口，本例中，主机是arete:40000。如果在修改配置之后，均衡器始终没有停止，你应该检查负责均衡的mongos的日志，查找错误。

在均衡器停止之后，就可以安全地开始备份了。备份完成后，不要忘了重新启动均衡器。为此，可以重新设置stopped的值：

```
> use config
> db.settings.update({_id: "balancer"}, {$set: {stopped: false}}, true);
```

为了简化与均衡器相关的一些操作，MongoDB v2.0引入了一些Shell辅助方法。例如，可以用sh.setBalancerState()来启动和停止均衡器：

```
> sh.setBalancerState(false)
```

这相当于调整settings集合中的stopped值。用这种方式禁用均衡器之后，可以不停地调用sh.isBalancerRunning()，直到均衡器停下为止。

7. 故障转移与恢复

虽然我们已经讲过了一般的副本集故障，但还是有必要提一下分片集群的潜在故障点和恢复的最佳实践。

- 分片成员故障

每个分片都由一个副本集组成。因此，如果这些副本集中的任一成员发生故障，从节点就会被选举为主节点，mongos进程会自动连接到该节点上。第8章描述了恢复副本集故障成员的具体步骤。选择哪种方法依赖于成员是何故障，但是不管怎么样，恢复的指南都是一样的，无论副本集是否是分片集群的组成部分。

如果发现副本集在故障转移之后有什么不正常的表现，可以通过重启所有mongos进程重置系统，这能保证适当连接都指向新的副本集。此外，如果发现均衡器不工作了，就检查config数据库的locks集合，找到process字段指向之前主节点的条目。如果有这样的条目，锁文档已经旧了，你可以安全地手动删除该文档。

- 配置服务器故障

一个分片集群要有三台配置服务器才能正常运作，其中最多能有两台发生故障。无论何时，

当配置服务器数量少于三台，剩余的配置服务器会变为只读状态，所有的拆分和均衡操作都会停止。请注意，这对整个集群没有负面影响，集群的读写仍能正常进行，当所有三台配置服务器都恢复之后，均衡器将从它停止的地方重新开始工作。

要恢复配置服务器，从现有的配置服务器把数据文件复制到发生故障的机器上，随后重启服务器。[①]

- mongos故障

要是mongos进程发生故障，没有什么好担心的。如果mongos运行在应用服务器上，它发生故障了，那么很有可能你的应用程序服务器也发生故障了。这时的恢复就是简单地恢复服务器。

无论mongos出于什么原因发生故障，进程本身都没有自己的状态。这意味着恢复mongos就是简单地重启进程，在配置服务器上指向它而已。

9.6　小结

分片是在大数据集下保持高读写性能的有效策略。MongoDB的分片能很好地工作于大量生产部署环境之中，也适用于你的情况。你可以充分利用MongoDB分片机制中现有的特性，不用自定义不成熟的分片解决方案。只要遵循本章的建议，特别是注意那些推荐的部署拓扑、选择分片键的策略，以及将数据保持在内存里的重要性，分片能让你获益匪浅。

① 和往常一样，在复制任何数据文件之前，确保已经锁定了mongod（第10章会做描述）或者正常关闭了该进程。不要在服务器仍在运行时复制任何数据文件。

第 10 章

部署与管理

10

本章内容

- ❑ 部署注意事项以及硬件要求
- ❑ 管理、备份与安全
- ❑ 性能调优

如果没有部署与管理相关的内容，本书就是不完整的。总而言之，使用MongoDB是一回事，而让它在生产环境中顺畅地运行则是另一回事。最后这一章的目标，就是让你在部署和管理MongoDB时能做出上佳的决策。你可以把本章看做是在提供宝贵的知识，以免你经历不愉快的生产数据库宕机。

开始时，我会讲述一些常见的部署问题，包括硬件要求、安全以及数据的导入导出。随后，本章罗列一些监控MongoDB的方法。我们还会讨论维护职责，其中最重要的是备份。我们将用常见的性能问题解决方案来结束本章。

10.1 部署

要成功部署MongoDB，你需要选择正确的硬件以及合适的服务器拓扑。如果有遗留数据，则需要知道如何才能有效地进行导入（和导出）。最后，还要确保你的部署是安全的。我们会在随后的小节中讨论这些问题。

10.1.1 部署环境

本节将介绍为MongoDB选择好的部署环境所要考虑的内容。我将讨论具体的硬件要求，例如CPU、内存和磁盘要求，为优化操作系统环境做些推荐，并提供一些关于云端部署的建议。

1. 架构

下面依次是两点与硬件架构有关的说明。

首先，因为MongoDB会将所有数据文件映射到一个虚拟的地址空间里，所以全部的生产部署都应该运行在64位的机器上。正如其他部分提到的那样，32位的架构会将MongoDB限制为仅有2 GB存储。开启了Journaling日志，该限制会减小到大约1.5 GB。这在生产环境里是很危险的，

因为如果超过这个限制，MongoDB的行为将无法预测。你可以随意使用32位机器进行单元测试和预发布，但在生产环境以及负载测试时，请严格使用64位架构。

其次，MongoDB必须运行于小端序（little-endian）机器上。这一点通常不难做到，但运行SPARC、PowerPC、PA-RISC以及其他大端序架构的用户就只能望洋兴叹了。[①]大多数的驱动同时支持小端与大端字节序，因此MongoDB的客户端通常在这两种架构上都能运行。

2. CPU

MongoDB并不是特别CPU密集型的；数据库操作很少是CPU密集型的。在优化MongoDB时，首要任务是确保该操作不是I/O密集型的（详见后续关于内存和磁盘的两部分内容）。

只有当索引和工作集都完全可放入内存时，你才可能遇到CPU的瓶颈。如果有一个MongoDB实例每秒钟处理成千上万（或数百）的查询，你能想到提供更多的CPU内核来提升性能。对于那些不使用JavaScript的读请求，MongoDB能够利用全部可用内核。

如果碰巧看到读请求造成CPU饱和，请检查日志中的慢查询警告。可能是缺少合适的索引，因此强制进行了表扫描。如果你开启了很多客户端，每个客户端都在运行表扫描，那么扫描加上它所带来的上下文切换会造成CPU饱和。这个问题的解决方案是增加必要的索引。

对于写请求，MongoDB一次只会用到一个核，这是由于全局写锁的缘故。因此扩展写负载的唯一方法是确保写操作不是I/O密集型的，并且用分片进行水平扩展。这个问题在MongoDB v2.0里有所好转，因为通常写操作不会在页错误时持有锁，而是允许完成其他操作。目前，有很多并发方面的优化正在开发之中，可能实现的几个选项是集合级锁（collection-level locking）和基于范围的锁（extent-based locking）。请查看JIRA和最新的发布说明以了解这些改进的开发状态。

3. 内存

和其他数据库一样，MongoDB在有大量内存时性能最好。请一定选择有足够内存的硬件（虚拟的或其他），足够容纳常用的索引和工作数据集。随着数据的增长，密切关注内存与工作数据集的比例。如果你让工作集大小超过内存，就可能看到明显的性能下降。从磁盘载入分页及分页这个过程本身不是问题，因为它是将数据载入内存的必要步骤。但是如果你对性能不满意，过多的分页可能就是问题所在。第7章详细讨论了工作集、索引大小和内存之间的关系。在本章末尾处，你会了解到识别内存不足的方法。

有一些情况下，你能安全地放任数据尺寸超出可用内存，但这仅仅是些例外，并非常见情况。一个例子是使用MongoDB进行归档，读和写都很少发生，并且不需要快速做出应答。在这种情况下，拥有和数据量一样的内存可能代价高昂却收效甚微，因为应用程序用不到那么多内存。对于完整的数据集，关键是测试。对应用程序的典型原型进行测试，确保能够得到所需的性能基线。

4. 磁盘

在选择磁盘时，你需要考虑IOPS（每秒的输入/输出操作数）以及寻道时间。这里不得不强调运行于单块消费级硬盘、云端虚拟盘（比方说EBS）以及高性能SAN之间的区别。一些应用程序在单块由网络连接的EBS卷上的性能还能接受，但是一些开销较大的应用程序就会有更高要求。

---

① 如果你对核心服务器的大端序支持感兴趣，请访问https://jira.mongodb.org/browse/SERVER-1625。

出于一些原因，磁盘性能是非常重要的。第一，在向MongoDB写入时，服务器默认每60 s与磁盘强制同步一次。这称为*后台刷新*（background flush）。对于写密集型应用与低速磁盘，后台刷新可能会因为速度慢对整体系统性能产生负面影响。第二，高速磁盘能让你更快地预热服务器。当需要重启服务器时，还要将数据集加载到内存里。这个过程是延时执行的；每次对MongoDB连续的读或写都会将一个新的虚拟内存页加载到内存里，直到物理内存被放满为止。高速磁盘能让这个过程执行得更快一些，这会提升MongoDB在冷重启之后的性能。最后，高速磁盘能改变应用程序工作集与所需内存的比例。比方说，相比其他磁盘，使用固态硬盘在运行时所需的内存更少（能有更大的容量）。

无论使用哪种类型的磁盘，通常在部署时都会比较严肃，不会只用一块硬盘，而是采用冗余磁盘阵列（RAID）。用户一般会使用Linux的LVM（Logical Volume Manager，逻辑卷管理器）来管理RAID集群，RAID级别[①]为10。RAID 10能在保持可接受性能的同时提供一定冗余，常用于MongoDB部署之中。

如果数据分散在同一台MongoDB服务器的多个数据库之中，还可以通过服务器的`--directoryperdb`标志确保它们的容量，这将在数据文件路径中为每个数据库创建单独的目录。有了它，你还可以为每个数据库挂载单独的卷（无论是否是RAID）。这能让你充分发挥各磁盘的性能，因为你可以从不同的磁盘组（或固态硬盘）上读取数据。

5. 文件系统

如果运行在正确的文件系统上，你将从MongoDB中获得最好的性能。特别是`ext4`和`xfs`这两个文件系统，提供了高速、连续的磁盘分配。使用这些文件系统能够提升常见的MongoDB预分配的速度。

一旦挂载了高速文件系统，还可以通过禁止修改文件的最后访问时间（`atime`）来提升性能。通常情况下，每次文件有读写时操作系统都会修改文件的`atime`。在数据库环境中，这带来了很多不必要的工作。在Linux上关闭`atime`相对比较容易。首先，备份文件系统的配置文件；然后，用你喜欢的编辑器打开原来的文件：

```
sudo cp /etc/fstab /etc/fstab.bak
sudo vim /etc/fstab
```

针对每个挂载的卷，你会看到一个列对齐的设置列表。在`options`列里，添加`noatime`指令：

```
# file-system        mount  type    options  dump  pass
UUID=8309beda-bf62-43  /ssd   ext4    noatime  0     2
```

保存修改内容，新的设置应该能立刻生效。

6. 文件描述符

一些Linux系统最多能打开1024个文件描述符。有时，这个限制对MongoDB而言太低了，在打开连接时会引起错误（在日志里能清楚地看到这点）。MongoDB顺理成章地要求每个打开的文件和网络连接都有一个文件描述符。假设将数据文件保存在一个文件夹里，其中有*data*这个单词，

① 如想对RAID级别有个概要认识，请访问http://en.wikipedia.org/wiki/Standard_RAID_levels。

可以通过lsof和一些管道看到数据文件描述符的数量：

```
lsof | grep mongo | grep data | wc -l
```

统计网络连接描述符数量的方法也很简单：

```
lsof | grep mongo | grep TCP | wc -l
```

对于文件描述符，最佳策略是一开始就设定一个很高的限额，使得在生产环境中永远都不会达到该值。可以使用ulimit工具检查当前的限额：

```
ulimit -Hn
```

要永久提升限额，可以用编辑器打开limits.conf：

```
sudo vim /etc/security/limits.conf
```

然后，设置软、硬限额，这些限额是基于每个用户指定的。下面的例子假设mongodb用户将运行mongod进程：

```
mongod soft nofile 2048
mongod hard nofile 10240
```

新设置将在用户下次登录时生效。

7. 时钟

事实证明，复制容易受到时钟偏移（clock skew）的影响。如果托管了副本集中多个节点的机器的时钟之间存在分歧，副本集就可能无法正常运作了。这可不是理想状态，幸好存在解决方案。你要确保每台服务器都使用了NTP（Network Time Protocol，网络时间协议），借此保持服务器时钟的同步。在Unix类的系统上，也就是运行ntpd守护进程；在Windows上，Windows Time Services就能担当这个角色。

8. 云

有越来越多的用户在虚拟化环境中运行MongoDB，这些环境统称为云。其中，亚马逊的EC2因其易用性、广泛的地理分布以及强有力的价格成为了用户的首选。EC2及其他类似的环境都能部署MongoDB，但你也要牢记它们的缺点，尤其是在应用程序要将MongoDB推向其极限之时。

EC2的第一个问题是你只能选择几种有限的实例类型。在本书编写时，还没有超过68 GB内存的虚拟实例。此类约束强迫你在工作集超过68 GB时对数据库进行分片，这并非适用于所有应用程序。如果能运行于真实的硬件之上，你可以拥有更多内存；相同的硬件成本下，这能影响分片的决定。

另一个潜在问题是EC2从本质上来说是一个黑盒，你可能会遭遇服务中断或实例变慢，却无法进行诊断或补救。

第三个问题与存储有关。EC2允许你挂载虚拟块设备，称为EBS卷。EBS卷提供了很大的灵活性，允许你按需添加存储并在机器间移动卷。它还能让你制作快照，以便用于备份。EBS卷的问题在于无法提供很高的吞吐量，尤其是在和物理磁盘进行比较时。为此，大多数MongoDB用户在EC2上托管重要应用程序时，都会对EBS做RAID 10，以此提升读吞吐量。这对高性能应用程序而言是必不可少的。

出于这些原因，比起处理EC2的限制和不可预测性，很多用户更青睐于在自己的物理硬件上运行MongoDB。但是，EC2和其他云环境非常方便，为很多用户所广泛接受。在正式使用云存储之前，要慎重考虑应用程序的情况，并在云中进行测试。

10.1.2 服务器配置

一旦决定了部署环境，你需要确定总体服务器配置。这涉及选择服务器的拓扑和决定是否使用Journaling日志，以及如何使用。

1. 选择一种拓扑结构

最小的推荐部署拓扑是三个成员的副本集。其中至少有两个是数据存储（非仲裁）副本，位于不同的机器上，第三个成员可以是另一个副本，也可以是仲裁节点。仲裁节点无需自己的机器，举例来说，你可以把它放在应用服务器上。第8章里有两套合理的副本集部署配置。

如果从一开始你就预计到工作集大小会超过内存，那开始时就可以使用分片集群了，其中至少包含两个副本集。第9章中有分片部署的详细推荐配置，还有关于何时开始分片的建议。

你可以部署单台服务器来支持测试和预发布环境。但对于生产环境部署而言，并不推荐采用单台服务器，就算开启了Journaling日志也是如此。只有一台服务器会为备份和恢复造成一定复杂性，当服务器发生故障时，无法进行故障转移。

但是，在极少的几种情况下也有例外。如果应用程序不需要高可用性或者快速恢复，数据集相对较小（比方说小于1 GB），那么运行在一台服务器上也是可以的。即使如此，考虑到日益下降的硬件成本，以及复制所带来的众多好处，先前提到的单机方案确实没什么亮点。

2. Journaling日志

MongoDB v1.8引入了Journaling日志，而MongoDB v2.0会默认开启Journaling日志。当Journaling日志开启时，MongoDB在写入核心数据文件时会先把所有写操作提交到Journaling日志文件里。这能让MongoDB服务器在发生非正常关闭时快速恢复并正常上线。

在v1.8之前没有此类特性，因此非正常关闭经常会导致灾难。这怎么可能呢？我之前多次提到MongoDB把每个数据文件映射到虚拟内存里。也就是说，当MongoDB执行写操作时，它是写入虚拟内存地址，而非直接写入磁盘。OS内核周期性地将这些写操作从虚拟内存同步到磁盘上，但是其频率和完整性是不确定的，因此MongoDB使用`fsync`系统调用每60 s对所有数据文件做一次强制同步。这里的问题在于，如果MongoDB进程在还有未同步的写操作时被杀掉了，那么则无法获知数据文件的状态。这就可能损坏数据文件。

在没有开启Journaling日志的`mongod`进程发生非正常关闭时，想将数据文件恢复到一致状态要运行一次修复。修复过程会重写数据文件，抛弃所有它无法识别的内容（损坏的数据）。因为大家通常都不太能接受停机和数据丢失，所以这种修复途径一般只能作为最后的恢复手段。从现有副本中重新同步数据几乎总是比较方便可靠的方法，这也是运行复制如此重要的原因之一。

Journaling日志让你不再需要修复数据库，因为MongoDB能用Journaling日志将数据文件恢复到一致状态。在MongoDB v2.0里，Journaling日志是默认开启的，但是你也可以通过–`nojournal`

标志禁用它：

```
$ mongod --nojournal
```

开启Journaling日志时，日志文件被放在一个名为journal的目录里，该目录位于主数据路径下面。

如果你在运行MongoDB服务器时开启了Journaling日志，请牢记几点。第一点，Journaling日志会降低写操作的性能。既想获得最高写入性能，又想有Journaling日志保障的用户有两个选择。其一，只在被动副本上开启Journaling日志，只要这些副本能和主节点保持一致，就无需牺牲性能。另一个解决方案，也许和前者是互补的，是为Journaling日志挂载一块单独的磁盘，然后在journal目录和辅助卷之间创建一个符号链接。辅助卷不用很大，一块120 GB的磁盘就足够了，这个大小的固态硬盘（SSD）的价格还是可以承受的。为Journaling日志挂载一块单独的SSD能确保将它运行时的性能损耗降到最小。

第二点，Journaling日志本身并不保证不会丢失写操作，它只能保证MongoDB始终能恢复到一致状态，重新上线。Journaling日志的机制是每100 ms将写缓冲和磁盘做一次同步。因此一次非正常关闭最多只会丢失100 ms里的写操作。如果你的应用程序无法接受这种风险，可以使用`getlasterror`命令的`j`选项，让服务器在Journaling日志同步后才返回：

```
db.runCommand({getlasterror: 1, j: true})
```

在应用程序层，可以用`safe`模式选项（与`w`和`wtimeout`类似）。在Ruby里，可以像这样使用`j`选项：

```
@collection.insert(doc, :safe => {:j => true})
```

一定要注意，每次写操作都像这样做是不明智的，因为这会强迫每次写操作都等到下次Journaling日志同步才返回。也就是说，所有的写操作都可能要等100 ms才能返回。因此请谨慎使用本特性。[①]

10.1.3　数据的导入与导出

如果你正从现有系统迁移到MongoDB，或者需要从数据仓库填充数据，那么就需要一种有效的导入方法。你可能还需要一个好的导出策略，因为可能要从MongoDB里将数据导出到外部处理任务中。例如，将数据导出到Hadoop进行批处理就已成为一种常见实践。[②]

有两种途径将数据导入和导出MongoDB，你可以使用自带的工具——`mongoimport`和`mongoexport`，或者使用某个驱动编写一个简单的程序。[③]

1. mongoimport与mongoexport

MongoDB自带了两个导入、导出数据的工具：`mongoimport`和`mongoexport`。你可以通过

① MongoDB的未来版本里会有更细粒度的Journaling日志同步控制，请查看最新的发布说明了解详细情况。

② 对于这种特定用法，在http://github.com/mongodb/mongo-hadoop可以找到官方支持的MongoDB-Hadoop适配器。

③ 注意，数据的导入和导出与备份有所不同，本章后面会讨论备份相关的内容。

mongoimport导入JSON、CSV和TSV文件，这通常用于从关系型数据库向MongoDB加载数据：

```
$ mongoimport -d stocks -c values --type csv --headerline stocks.csv
```

本例中，你将一个名为stocks.csv的CSV文件导入到了stocks数据库的values集合里。--headerline标志表明了CSV的第一行包含字段名。可以通过mongoimport --help看到所有的导入选项。

可以通过mongoexport将一个集合的所有数据导出到一个JSON或CSV文件里：

```
$ mongoexport -d stocks -c values -o stocks.csv
```

这条命令会将数据导出到stocks.csv文件里。与mongoimport类似，你可以通过--help看到mongoexport的其他命令选项。

2. 自定义导入与导出脚本

当处理的数据相对扁平时，你可能会使用MongoDB的导入导出工具；一旦引入了子文档和数组，CSV格式就有些“力不从心”了，因为它不是设计来表示内嵌数据的。当需要将富文档导出到CSV或者从CSV导入一个富MongoDB文档，也许构建一个自定义工具会更方便。你可以使用任意驱动实现这一目标。例如，MongoDB用户通常会编写一些脚本连接关系型数据库，随后将两张表的数据整合到一个集合里。

将数据移入和移出MongoDB是件很复杂的事：数据建模的方式会因系统而异。在这些情况下，要做好将驱动当成转换工具的准备。

10.1.4 安全

大多数RDBMS都有一套复杂的安全子系统，可以对用户和用户组授权，进行细粒度的权限控制。与此相反，MongoDB v2.0只支持简单的、针对每个数据库的授权机制。这就让运行MongoDB的机器的安全性变得更加重要了。此处我们会讨论在安全环境里运行MongoDB所需考虑的一些重点内容，并解释身份验证是如何进行的。

1. 安全环境

和所有数据库一样，MongoDB应该运行在一个安全环境里。生产环境中，MongoDB的用户必须利用现代操作系统的安全特性来确保数据的安全。在这些特性之中，也许最重要的就是防火墙了。在结合使用防火墙与MongoDB时，唯一潜在的难点是了解哪台机器需要和其他机器互相通信。还好，通信规则很简单。在副本集中，每个节点都要能和其他节点通信。此外，所有数据库客户端都必须能连接到各个它可能会通信的副本集节点上。

分片集群中含有副本集，因此所有副本集的规则都能适用；在分片的情况下，客户端是mongos路由器。除此之外：

- 所有分片都必须能与其他分片直接通信；
- 分片与mongos路由器都必须能连上配置服务器。

相关的安全关注点是*绑定地址*（bind address）。默认情况下，MongoDB会监听本机的所有地址，但你可能只想监听一个或几个特殊的地址。为此，可以在启动mongod和mongos时带上

--bind_ip选项，它接受一个或多个逗号分隔的IP地址。例如，想要监听loopback接口和内部IP地址10.4.1.55，可以像这样启动mongod：

```
mongod --bind_ip 127.0.0.1,10.4.1.55
```

请注意，机器之间发送数据都使用明文，官方的SSL支持安排在MongoDB v2.2中发布。

2. 身份验证

MongoDB的身份验证最早是为那些在共享环境下托管MongoDB服务器的用户构建的。它的功能并不多，但在需要一些额外安全保障时还是很有用的。我们先讨论一下身份验证API，然后再描述如何在副本集和分片中使用该API。

● 身份验证API

要着手了解身份验证，先创建一个管理员用户，切换到admin数据库，运行db.addUser()，该方法接受两个参数：一个用户名和一个密码。

```
> use admin
> db.addUser("boss", "supersecret")
```

管理员用户能创建其他用户，还能访问服务器上的所有数据库。有了它，你就能开启身份验证了，在重启mongod实例时加上--auth选项：

```
$ mongod --auth
```

现在，只有通过身份验证的用户才能访问数据库。重启Shell，随后使用db.auth()方法以管理员身份登录：

```
> use admin
> db.auth("boss", "supersecret")
```

现在可以为个别数据库创建用户了。如果想要创建只读用户，将true作为db.addUser()方法的最后一个参数。这里将为stocks数据库添加两个用户。第一个用户拥有所有权限，第二个只能读取数据库的数据：

```
> use stocks
> db.addUser("trader", "moneyfornuthin")
> db.addUser("read-only-trader", "foobar", true)
```

现在，只有三个用户能访问stocks数据库，他们是boss、trader和read-only-trader。如果你希望查看拥有某个数据库访问权限的所有用户的列表，可以查询system.users集合：

```
> db.system.users.find()
{ "_id" : ObjectId("4d82100a6dfa7bb906bc4df7"),
  "user" : "trader", "readOnly" : false,
  "pwd" : "e9ee53b89ef976c7d48bb3d4ea4bffc1" }
{ "_id" : ObjectId("4d8210176dfa7bb906bc4df8"),
  "user" : "read-only-trader", "readOnly" : true,
  "pwd" : "c335fd71fb5143d39698baab3fdc2b31" }
```

从该集合中删除某个用户，就能撤销它对某个数据库的访问权限。如果你更青睐于辅助方法，可以使用Shell里的db.removeUser()方法，它的作用是一样的。

你并不需要显式注销，中断连接（关闭Shell）就行了。但是如果你需要，也有注销命令可用：

```
> db.runCommand({logout: 1})
```

当然，你也可以通过驱动使用我们此处看到的全部身份验证逻辑，请查看驱动的API了解更多详情。

● 副本集身份验证

副本集也支持刚才介绍的身份验证API，但是为副本集开启身份验证还需要额外的几个步骤。开始时，创建一个文件，其中至少包含6个Base64字符集[1]中的字符。文件的内容会被作为某种密码，每个副本集成员都会用它来和其他成员进行身份验证。举个例子，你可以创建一个名为secret.txt的文件，其内容如下：

```
tOps3cr3tpa55word
```

将该文件放到每个副本集成员的机器上，调整文件权限，以便只有文件的拥有者才能访问它：

```
sudo chmod 600 /home/mongodb/secret.txt
```

最后，在启动每个副本集成员时使用--keyFile选项指定密码文件的位置：

```
mongod --keyFile /home/mongodb/secret.txt
```

现在副本集就已经开启身份验证了，你会希望事先创建一个管理员用户，就像上一节里那样。

● 分片身份验证

分片身份验证是副本集身份验证的一个扩展。集群里的每个副本集都已经像刚才介绍的那样，通过密钥文件保护起来了。此外，所有的配置服务器和每个mongos实例也都拥有一个包含相同密码的密钥文件。在启动每个进程时都用--keyFile选项指定包含密码的文件，整个分片集群都使用该密码。完成这个步骤，整个集群就能使用身份验证了。

10.2 监控与诊断

在生产环境中部署完MongoDB，你就希望能了解它的运转情况。如果系统性能在慢慢下降或者经常发生故障，你希望能够知道这些情况，这就该用到监控了。让我们先从最简单的监控开始：日志。随后我们会看一些内置命令，它们能提供正在运行的MongoDB服务器的大多数信息；这些命令是mongostat工具和Web控制台的基础，我会对mongostat工具和Web控制台做个简要说明。我还会推荐几个外部监控工具。本节最后会介绍两个诊断工具：bsondump和mongosniff。

10.2.1 日志

日志是第一级监控，正因如此，你应该计划保留所有部署的MongoDB的日志。[2]通常这都不是问题，因为MongoDB在后台运行时要求你指定--logpath选项。此外，还有一些需要留意的额外设置。要开启详细日志（verbose logging），在启动mongod进程时加上-vvvvv选项（v越多，输出越详细）。举例来说，如果需要调试一些代码，想要在日志里记录下每个查询，这就很方便。但是也要注意，详细日志会让日志文件变得很大，可能会影响服务器的性能。

① Base64字符集由以下字符组成：英文字母中的全部大写和小写字母、数字0～9以及+和/。

② 不要简单地通过管道将日志输出到/dev/null或stdout。

其次，可以在启动`mongod`时使用`--logappend`选项，这会让日志追加到现有日志文件后面，而非覆盖它。

最后，如果有一个长时间运行的MongoDB进程，你可能想写一个脚本周期性地滚动日志文件，为此，MongoDB提供了`logrotate`命令，可以像这样在Shell里运行该命令：

```
> use admin
> db.runCommand({logrotate: 1})
```

向进程发送`SIGUSR1`信号也能运行`logrotate`命令，下面是如何向进程号为12345的进程发送`SIGUSR1`信号：

```
$ kill -SIGUSR1 12345
```

10.2.2　监控工具

本节我会介绍MongoDB自带的监控命令和工具。

1. 数据库命令

有三个展示MongoDB内部状态的数据库命令，它们是所有MongoDB监控应用程序的基础。

- `serverStatus`

`serverStatus`命令的输出真是名副其实的内容丰富。统计的所有信息当中包含页错误、B树访问率、打开连接数，以及总的插入、更新、查询和删除。下面是一段节选后的`serverStatus`命令输出：

```
> use admin
> db.runCommand({serverStatus: 1})
{
  "host" : "ubuntu",
  "version" : "1.8.0",
  "process" : "mongod",
  "uptime" : 246562,
  "localTime" : ISODate("2011-03-13T17:01:37.189Z"),

  "globalLock" : {
    "totalTime" : 246561699894,
    "lockTime" : 243,
    "ratio" : 9.855545289656455e-10,
    "currentQueue" : {
      "total" : 0,
      "readers" : 0,
      "writers" : 0
    },
  },
  "mem" : {
    "bits" : 64,
    "resident" : 3580,
    "virtual" : 9000,
    "mapped" : 6591
  }

"ok" : 1 }
```

globalLock部分很重要，因为它揭示了服务器花在写锁上的总时间。这里的高比例说明写操作有瓶颈。currentQueue也许是更具体的瓶颈表述，如果有大量的读或写等在队列里，那么就该进行某种优化了。

mem部分显示了mongod进程是如何使用内存的。bits字段说明这是一台64位的机器。resident是MongoDB所占用的物理内存数量。virtual是进程所映射的虚拟内存的兆字节数，mapped是virtual的子集，标明那些内存里有多少是只用来映射数据文件的。本例中，有大约6.5 GB的数据文件被映射到了虚拟内存里，其中3.5 GB是物理内存。我反复强调，理想情况下工作集应该能被放到内存里，mem部分能提供一个大概的信息，说明情况是否如此。

每个版本的MongoDB里，serverStatus的输出都会有所变化并得以改进，因此像本书这样在非永久性媒介里为该命令编写文档并不总是很有帮助。你可以在http://www.mongodb.org/display/DOCS/serverStatus看到该命令的最新详细说明。

- top

top命令会显示每个数据库的操作计数器。如果应用程序使用了多个物理数据库，或者你想看看操作的平均耗时，那么这是个有用的命令。下面是一些示例输出：

```
> use admin
> db.runCommand({top: 1}) {
"totals" : { "cloud-docs" :
{ "total" : { "time" : 194470, "count" : 20 },
  "readLock" : { "time" : 324, "count" : 12 },
  "writeLock" : { "time" : 194146, "count" : 8 },
  "queries" : { "time" : 194470, "count" : 20 },
  "getmore" : { "time" : 0, "count" : 0 } },
"ok" : 1}
```

此处可以看到很多时间都花在了写锁上，值得深入调查一下，看看写操作是否有可以优化的地方。

- db.currentOp()

能知道MongoDB目前正在做什么常常很有用，db.currentOp()方法就能揭示这个信息，它会返回当前正在运行的所有操作，以及正在等待运行的其他操作。下面是该方法的输出示例，它是在上一章里配置的分片集群上运行的：

```
db.currentOp()
[ {
      "opid" : "shard-1-test-rs:1232866",
      "active" : true,
      "lockType" : "read",
      "waitingForLock" : false,
      "secs_running" : 11,
      "op" : "query",
      "ns" : "docs.foo",
      "query" : {
        "$where" : "this.n > 1000"
      },
      "client_s" : "127.0.0.1:38068",
      "desc" : "conn"
    } ]
```

现在正在执行一条特别慢的查询，可以看到它已经运行了11 s了，和所有查询一样，它会占用读锁。如果这个操作有问题，你可能会想调查一下它的调用源，可以看看client字段。啊呀，这是一个分片集群，因此调用源是mongos进程，正如client_s字段名所标识的那样。如果要杀掉这个操作，可以将opid传给db.killOp()方法：

```
db.killOp("shard-1-test-rs:1232866")
{
  "op" : "shard-1-test-rs:1233339",
  "shard" : "shard-1-test-rs",
  "shardid" : 1233339
}
```

如果想要查看当前MongoDB服务器上正在运行的所有操作的列表，可以使用如下虚拟命令：

```
db['$cmd.sys.inprog'].find({$all: 1})
```

2. mongostat

db.currentOp()方法只会显示特定时刻排在队列中或者正在执行的操作。类似的，serverStatus命令只提供某一时间点上不同系统字段和计数器的快照。但是，有些时候你需要系统实时活动的视图，这时就该mongostat登场了。mongostat效仿iostat和其他类似的工具，以固定时间间隔查询服务器信息，显示统计数据的矩阵，从每秒插入数到常驻内存量，再到B树页丢失频率。

可以在localhost上调用mongostat命令，显示信息每秒滚动一次：

```
$ mongostat
```

mongostat命令同样也是高度可配置的，可以通过--help查看所有选项。它还有一个更出名的特性，即集群发现（cluster discovery）；在启动mongostat时带上--discover选项，你可以将它指向单个节点，它会发现副本集或分片集群中的剩余节点，随后聚合显示整个集群的统计信息。

3. Web控制台

通过Web控制台，你能以更可视化的方式获得某个运行中的mongod进程的信息。每个mongod进程都会监听服务器端口往上第1000个端口的HTTP请求。如果你的mongod运行在27017端口，那么Web控制台就在28017端口。如果运行在localhost上，可以将Web浏览器指向http://localhost:28017，你会看到如图10-1所示的页面。

开启服务器的基本REST接口后，还能获得更多状态信息。如果在启动mongod时加上--rest，就能开启很多额外的Web控制台命令，Web控制台的登录页面上有指向它们的链接。

mongod arete.local

List all commands | Replica set status

Commands: buildInfo cursorInfo features isMaster listDatabases replSetGetStatus serverStatus top

```
db version v1.8.0-rc0, pdfile version 4.5
git hash: 65a7e81df0747b6bc9380b78e0182192bacdb4d0
sys info: Darwin arete.local 10.6.0 Darwin Kernel Version 10.6.0: Wed Nov 10 18:13:17 PST 2010; root:xnu-1504.9.26~3/R
uptime: 170 seconds
```

low level requires read lock

```
time to get readlock: 0ms
# databases: 1

replication:
master: 0
slave:  0
initialSyncCompleted: 1
```

clients

Client	OpId	Active	LockType	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
initandlisten	0		W			2004	webinar	{ name: /^local.temp./ }	0.0.0.0:0		
snapshotthread	0		0			0			(NONE)		
clientcursormon	0		R			0			(NONE)		
websvr	0		0			0			(NONE)		

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	13	7.1%	1	0.0%	12	7.1%	13	7.1%	0	0%	0	0%	0	0%	0	0%
a	1	0.6%	0	0%	1	0.6%	1	0.6%	0	0%	0	0%	0	0%	0	0%
app	1	1.0%	0	0%	1	1.0%	1	1.0%	0	0%	0	0%	0	0%	0	0%
crawler	1	0.3%	0	0%	1	0.3%	1	0.3%	0	0%	0	0%	0	0%	0	0%
foo	1	0.9%	0	0%	1	0.9%	1	0.9%	0	0%	0	0%	0	0%	0	0%

图10-1　MongoDB Web控制台

10.2.3　外部监控应用程序

大多数重要的部署都要求有外部监控应用，Nagios和Munin是两款流行的开源监控系统，很多MongoDB部署都用它们来进行监控。两款工具都只需安装一个简单的开源插件就能监控MongoDB。

编写一个针对某款监控应用程序的插件并非难事，一般都涉及针对某个在线MongoDB数据库运行不同统计命令。`serverStatus`、`dbstats`和`collstats`命令通常就能提供需要的所有信息，你能直接通过HTTP REST接口获得所有这些信息，不需要使用驱动。

10.2.4　诊断工具（mongosniff、bsondump）

MongoDB包含两个诊断工具。第一个是`mongosniff`，它能侦听客户端发给MongoDB服务器的数据包并将其以易于理解的形式输出。如果恰好要编写一个驱动或是调试一个错误连接，那这就是最好的工具。可以像下面这样启动`mongosniff`，监听本地网络接口的默认端口：

```
sudo mongosniff --source NET I0
```

有客户端（比方说MongoDB Shell）连接上来之后，你会得到一个简单易读的网络交互情况：

```
127.0.0.1:58022  -->> 127.0.0.1:27017 test.$cmd 61 bytes
  id:89ac9c1d  2309790749 query: { isMaster: 1.0 }  ntoreturn: -1
127.0.0.1:27017  <<--  127.0.0.1:58022   87 bytes
  reply n:1 cursorId: 0 { ismaster: true, ok: 1.0 }
```

通过`--help`可以看到`mongosniff`的所有选项。

另一个有用的工具是`bsondump`，允许你查看原始BSON文件。BSON文件是由`mongodump`工具（稍后会讨论它）和副本集回滚来生成的。[①]举例来说，假设你导出了一个只有单个文档的集合。如果那个集合最终被放到一个名为`users.bson`的文件里，那么可以轻松地用如下命令查看文件内容：

```
$ bsondump users.bson
{ "_id" : ObjectId( "4d82836dc3efdb9915012b91" ), "name" : "Kyle" }
```

可以看到，`bsondump`默认将BSON输出为JSON。如果正进行重要的调试工作，你需要查看真正的BSON类型构成及大小。为此，以调试模式运行工具：

```
$ bsondump --type=debug users.bson
--- new object ---
   size : 37
     _id
       type:  7 size: 17
     name
       type:  2 size: 15
```

该命令显示了对象的总大小（37字节）、两个字段的类型（7和2）以及那些字段的大小。

10.3　维护

本节我会介绍三个最常用的MongoDB维护任务，首先要讨论的是备份。和其他数据库一样，你也该有个日常备份策略。随后，我会介绍压紧（compaction），因为在少数几种情况下，数据文件需要压紧。最后再简要地说一下升级，在条件允许时，你会希望运行最新的稳定版MongoDB。

10.3.1　备份与恢复

在运行生产环境数据库时，有一部分工作内容就是准备应对灾难，备份在其中扮演了重要的角色。当灾难不期而至时，好的备份能力挽狂澜，这时你绝不会为日常备份所投入的时间和精力而感到后悔。但还是有些用户决定不做备份，当他们遇到问题无法恢复自己的数据库时，只能说是自作自受，你可千万别向他们学习。

---

① 还有其他一些情况下你也会看到原始BSON文件，但是MongoDB的数据文件并非其中之一，所以不要尝试用`bsondump`查看它们。

MongoDB数据库有两个常规的备份策略，第一个是使用mongodump和mongorestore工具；第二，而且很可能是更常用的，是复制原始的数据文件。

1. mongodump与mongorestore

mongodump能把数据库的内容导出成BSON文件，而mongorestore则能读取并还原这些文件。这些工具在备份单个集合、数据库乃至整个服务器时都非常有用。它们能运行于线上服务器（无需锁定或关闭服务器），你也可以在服务器被锁定或关闭时将它们指向一组数据文件。最简单的mongodump运行方法如下：

```
$ mongodump -h localhost --port 27017
```

这能把localhost服务器上的每个数据库和集合都导出到名为dump的目录里。导出的内容包含每个集合里的所有文档，还包含定义了用户和索引的系统集合。但值得注意的是索引本身并不包含其中，也就是说，在恢复时必须重建全部索引。如果你的数据集特别大，或是拥有大量索引，那么这会花费不少时间。

在还原BSON文件时，运行mongorestore，将它指向dump文件夹：

```
$ mongorestore -h localhost --port 27017 dump
```

请注意，在还原过程中mongorestore默认不会删除数据。因此，如果你向一个现有数据库还原数据，请务必带上--drop标志。

2. 基于数据文件的备份

大多数用户选择基于文件的备份，将原始数据文件复制到一个新的位置。这种方法在大多数情况下比mongodump要快，因为在备份和还原时无需转换数据。[①]唯一潜在的问题是基于文件的备份要求锁定数据库，但是通常你都会锁定从节点，因此在备份的过程中应用程序应该能够保持在线。

复制数据文件

用户经常会犯错误，没有先锁定数据库就去复制数据文件或制作快照。在禁用Journaling日志时，这会造成数据文件损坏。在开启Journaling日志时，制作快照没问题，但复制数据文件有点麻烦，容易发生状况。

因此，无论是否开启了Journaling日志，本书建议总是在复制数据文件或制作磁盘快照前锁定数据库。比起锁定数据库所带来的安宁和对文件完整性的保障，由此引发的轻微延时是值得的。

复制数据文件，先要确认它们都处于一致状态，为此可以关闭数据库或是锁定它。由于关闭数据库在一些部署情况下太麻烦了，所以大多数用户都选择进行锁定。以下是用来同步并锁定数据库的命令：

```
> use admin
> db.runCommand({fsync: 1, lock: true})
```

① 举个例子，采用这种策略会保留全部的索引——无需在还原时重建索引。

此时，数据库是写锁定的，数据文件都同步到了磁盘上，也就是说可以安全地复制数据文件了。如果正运行在一个支持快照的文件系统或存储系统上，最好先制作一个快照，以后再做复制，这能让你快点解锁。

如果无法制作快照，就必须在复制数据文件时让数据库保持在锁定状态。如果是从一个从节点复制数据文件，请确保该节点仍连着主节点，并有足够的oplog让它在备份期间保持离线状态。

一旦完成快照或者备份，就可以解锁数据库了。看似神秘的解锁命令是这样的：

```
> db.$cmd.sys.unlock.findOne()
> { "ok" : 1, "info" : "unlock requested" }
```

请注意，这仅仅是请求解锁，数据库可能不会立刻解锁，可以运行`db.currentOp()`方法验证数据库是否已经解锁。

10.3.2　压紧与修复

MongoDB包含了修复数据库的功能，可以通过命令行触发修复服务器上的所有数据库：

```
$ mongod --repair
```

也可以运行`repairDatabase`命令修复单个数据库：

```
> use cloud-docs
> db.runCommand({repairDatabase: 1})
```

修复是个离线操作，在执行时，数据库的读写都将被锁定。修复就是读取和重写所有数据文件并重建各个索引，在此过程中丢弃掉损坏的文档。也就是说要修复数据库，需要有足够的空余磁盘空间来存储重写的数据。要说修复的开销很大，那还是轻的，修复大型数据库能花好几天。

MongoDB的修复最初是用作恢复受损数据库的最后一道防线。在未正常关闭，又没有开启Journaling日志时，修复是让数据文件回到一致状态的唯一途径。幸运的是，如果部署时使用了复制，至少有一台机器开启了Journaling日志，并且进行日常线下备份，你应该永远也用不上执行修复的恢复功能。依靠修复来进行恢复是种愚蠢的做法，应尽量避免这么做。

那么数据库修复又有什么好处呢？运行修复能压紧数据文件并重建索引。自v2.0版本起，MongoDB对数据文件压紧并没有太好的支持，因此如果执行了很多随机删除，尤其是删除小文档（小于4 KB），那么总存储大小可能仍然保持不变甚至是继续增长。压紧数据文件能有效应对此类对空间的过度使用。

要是没有时间或资源执行完整的修复，还有两个选择，它们都是针对单个集合进行操作的：可以重建索引或是压紧集合。要重建索引，可以使用`reIndex()`方法：

```
> use cloud-docs
> db.spreadsheets.reIndex()
```

这招很管用，但一般而言，索引空间是能高效重用的；数据文件空间才是问题。所以`compact`命令通常是更好的选择。`compact`会重写数据文件，并重建集合的全部索引。下面展示如何在Shell中运行`compact`命令：

```
> db.runCommand({ compact: "spreadsheets" })
```

该命令的初衷是在运行中的从节点上压紧，以此避免停机时间。一旦完成副本集中所有从节点的压紧，可以降级主节点，再对它进行压紧。如果必须在主节点上运行compact命令，可以向命令键中添加{force: true}。请注意，如果选择这种方式，该命令会对系统进行写锁定：

```
> db.runCommand({ compact: "spreadsheets", force: true })
```

10.3.3　升级

MongoDB还是一个相对比较“年轻”的项目，这意味着它的新版本中一般都包含很多重要的补丁和性能改进。出于这些原因，你应该尽可能运行最新的稳定版本。至少到v2.0为止，升级的过程就是简单地关闭老的mongod进程，用老的数据文件启动新的mongod进程。MongoDB的后续版本可能会对索引和数据文件格式做些小的改变，这可能会让升级过程稍微烦琐一点。请查看最新的发布说明以了解正确的推荐做法。

当然，在升级MongoDB时，可能会要升级副本集集群，这种情况下，常规的策略是一次升级一个节点，先从从节点开始。

10.4　性能调优

最后这一节里，我会提几个与诊断和解决性能问题相关的内容。

大多数MongoDB的性能问题都能追溯到一个源头：硬盘。本质上来看，对磁盘施加的压力越大，MongoDB就运行得越慢，因此大多数性能优化的目标就是减少对磁盘的依赖。有多种方式能达到这个目标，但在我们深入了解它们之前，先了解如何弄清磁盘性能还是很有必要的。在Unix类系统上，iostat就是一款理想的工具。以下示例中，我通过-x选项显示扩展统计信息，2说明以2 s为间隔进行显示：[1]

```
$ iostat -x 2
Device:  rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb       0.00   3101.12    10.09    32.83  101.39   1.34  29.36

Device:  rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb       0.00   2933.93     9.87    23.72  125.23   1.47  34.13
```

想要详细了解每一个字段的含义，请查看系统的man页面。要快速诊断问题，你会对最后三列最感兴趣。await以毫秒为单位，标明了处理I/O请求的平均时间，该平均值包含了I/O队列里的时间和实际处理I/O请求的时间。svctime标明了处理请求所花费的平均时间。%util则是CPU用来处理磁盘I/O请求所耗时间的百分比。

之前的iostat片段显示了一个中等程度的磁盘占用情况。I/O的平均等待时间大约是100 ms（提示：这已经不少了！），平均处理时间大约是1 ms，利用率大约是30%。如果查看该机上的MongoDB日志，你能看到很多慢操作（查询、插入或其他操作）。事实上，正是那些慢操作最早让你对潜在的问题有所警觉。iostat的输出能帮助你确认问题。请注意，MongoDB系统的磁盘

① 注意，该示例是针对Linux的；在Mac OS X上，对应命令是iostat -w 2。

利用率达到100%是很常见的；虽然仅由MongoDB的问题造成磁盘利用率高的情况很少，但这些用户还是觉得被MongoDB折腾得够呛。在后续的五个小节里，我会给出一些优化数据库操作、减轻磁盘负载的方法。

10.4.1　为提升性能检查索引和查询

发现性能问题时，应该首先检查索引。这里假设应用程序会发起查询和更新，它们是使用索引的主要操作。[1]第7章大致罗列了识别并修复慢操作的步骤；其中涉及开启查询剖析器，随后确保每个查询与更新都能有效利用索引。总的来说，这意味着每个操作要扫描尽可能少的文档。

保证没有冗余的索引也同样重要，因为冗余索引会占用磁盘空间、消耗更多内存，在每次写入时还需要做更多工作。第7章中有消除此类冗余索引的方法。

然后呢？在审视了索引和查询之后，你会发现效率低下的部分，把性能问题一起修正掉。日志里再也看不到慢查询警告了，iostat的输出也显示利用率下降了。调整索引在修正性能问题方面的效果比你想的要好；在发现性能问题时，索引应该是你首先检查的地方。

10.4.2　添加内存

修改索引并非总是有效，也许你已经有了最优化的查询，有了完美的索引，但是磁盘利用率还是居高不下。此时，你应该看看索引尺寸和工作集对物理内存的比例。在应用程序所用到的每个数据库上运行stats()命令：

```
> use app
> db.stats()
{
    "db" : "app",
    "collections" : 5,
    "objects" : 3932487,
    "avgObjSize" : 543.012,
    "dataSize" : 2135390324,
    "storageSize" : 2419106304,
    "numExtents" : 38,
    "indexes" : 4,
    "indexSize" : 226608064,
    "fileSize" : 6373244928,
    "nsSizeMB" : 16,
    "ok" : 1
}
```

现在看一下数据和索引的大小，数据大小刚超过2 GB，索引大小大约是230 MB。假设工作集包含了系统中的全部数据，那么机器上至少要有3 GB内存才能避免频繁访问磁盘。要是这台机器只有1.5 GB内存，那你就只能眼睁睁看着磁盘利用率居高不下了。

在查看数据库统计信息时，还要注意dataSize和storageSize之间的区别。当storageSize

---

① 某些数据库命令，比如count，同样也会使用索引。

超过`dataSize`两倍以上时，就会因磁盘碎片而影响性能。这些碎片会迫使服务器使用更多的内存；遇到这种情况，在添加更多的内存之前，应该先尝试压紧数据文件。请查看本章之前与压紧有关的内容，了解如何进行压紧。

10.4.3　提升磁盘性能

添加内存的方法也有一些问题。首先，并非总能添加内存，例如，你正在使用EC2，虚拟机的最大可用内存上限就是68 GB。其次，添加内存并非总能解决I/O问题。举例来说，如果应用程序是写密集型的，后台的刷新或者向内存加载新数据分页依然会压垮你的磁盘。所以说，如果有了高效的索引和充足的内存，但还是觉得磁盘I/O缓慢，就该想想提升磁盘性能了。

提升磁盘性能有两种方法。一种是购买更快的磁盘，买块15 K RPM的硬盘或者SSD还算是笔划算的投入。除此之外，把磁盘配置成RAID阵列，也能够提升读写的吞吐量，这种方式也可以作为前者的补充。[①]如果配置得当，RAID阵列也许可以解决I/O瓶颈。正如之前所说的那样，在EBS卷上运行RAID 10能显著提升读取的吞吐量。

10.4.4　水平扩展

在解决性能问题时，下一步就该用到水平扩展了。你有两条路可选。如果应用程序是读密集型的，单个节点可能无法应对所有查询，即使内存中是优化后的索引和数据。这时可以让读操作分布在各个副本上。官方的MongoDB驱动支持跨副本集成员进行操作，在逐步提升到分片集群前，这个策略值得一试。

当所有其他手段都无效之时，就得进行分片了。满足以下条件时，你就应该转向分片集群了：

- 无法将工作集完整加载到任意一台机器的物理内存里；
- 对任意服务器而言，写负载太密集了。

要是在配置了分片集群之后，还是存在性能问题，那你就该回过头去，确保所有索引都经过了优化、数据都在内存里，并且磁盘性能良好。要获得最佳的硬件利用率，你可能需要添加更多分片。

10.4.5　寻求专业帮助

造成性能下降的原因多种多样，并且经常很奇特。从糟糕的Schema设计到诡异的服务器缺陷都能影响性能。如果在尝试了各种可能的办法之后，仍然无法解决问题，你应该考虑让对MongoDB更有经验的人士来检查你的系统。一本书的作用有限，但是经验丰富的专业人士所发挥的作用就大不相同了。当你不知所措或者心存疑虑时，请寻求专业帮助。性能问题的解决方案有时完全是凭直觉的。

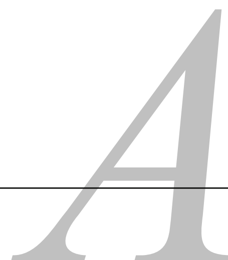
① RAID还有另一个好处，即在恰当的RAID级别下，能够获得磁盘冗余。

10.5　小结

本章讲述了在生产环境部署MongoDB时最重要的一些考量点。你现在应该已经有了足够的知识选择合适的硬件、监控系统运行并进行日常备份。此外，你还了解了如何解决性能问题。最终，这些知识都会随着经验一同成长。除了那些无法预见的情况，我相信MongoDB经得起本章讲到的这些问题的考验。MongoDB努力让生活更简单，但是数据库和它们与应用程序的交互实在是太复杂了。要是本书的建议没能奏效，知识渊博的专家一定对你大有帮助。

附　录

安　　装



在附录A中，你将了解到如何在Linux、Mac OS X和Windows上安装MongoDB，对常用的配置选项有个大概的认识。针对开发者，会有一些从源代码编译MongoDB的说明。

结尾时我会简述Ruby和RubyGems的安装，这能对希望运行书中那些Ruby示例的读者有所帮助。

A.1　安装

在给出安装指南之前，先说明一下MongoDB的版本号。简而言之，你应该运行最新的稳定版本。MongoDB的稳定版用偶数次版本号来标记，因此：1.8、2.0和2.2这些版本是稳定版；1.9和2.1是开发版，不应该使用在生产环境里。http://mongodb.org上的下载页面提供了针对32位和64位系统编译的静态链接二进制包，其中可以找到最新的稳定版本、开发分支以及最新修改的每日构建版本。在大多数平台上，这些二进制包是安装MongoDB的最简单途径，包括Linux、Mac OS X、Windows和Solaris，这也是我们所推崇的安装方法。

A.1.1　在Linux上安装MongoDB

有三种方式在Linux上安装MongoDB，你可以从mongodb.org网站直接下载预先编译好的二进制包、使用包管理器或者是用源代码手动编译。随后的小节中我们会讨论头两种方法，随后再提供一些编译相关的说明。

1. 使用预编译的二进制包进行安装

先访问http://www.mongodb.org/downloads，你能看到所有可供下载的最新MongoDB二进制包，选择适用于你系统的最新稳定版本的下载URL。以下示例使用针对64位系统编译的MongoDB v2.0。

使用浏览器或curl工具下载压缩包，随后通过tar解压：

```
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.0.0.tgz
    > mongo.tgz
$ tar -xzvf mongo.tgz
```

要运行MongoDB，你需要一个数据目录。mongod守护进程默认会把它的数据文件保存在

/data/db之中。创建该目录，确保它有合适的权限：

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

你已经可以启动服务器了，切换到MongoDB的`bin`目录，运行`mongod`可执行文件：

```
cd mongodb-linux-x86_64-2.0.0/bin
./mongod
```

要是一切顺利，应该可以看到一些和以下启动日志（经删减）类似的东西。请注意最后几行，确认服务器正监听默认的27017端口：

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
  pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

如果服务器意外终止，请阅读A.5节。

2. 使用包管理器

包管理器能极大简化MongoDB的安装，唯一的主要劣势是包维护方可能无法始终跟上最新的MongoDB版本。运行最新的稳定版本是很重要的，因此如果选择使用包管理器进行安装，请确保正在安装的MongoDB是最近的版本。

如果恰巧运行在Debian、Ubuntu、CentOS或Fedora上，那么始终可以安装最新版本。因为10gen为这些平台维护并发布自己的包。在mongodb.org网站上可以找到安装这些特殊包的更多信息。在http://mng.bz/ZffG可以找到针对Debian和Ubuntu的指南；至于CentOS和Fedora，请访问http://mng.bz/JSjC。

同样也有用于FreeBSD和ArchLinux的包，请查看它们对应的包仓库。

A.1.2 在Mac OS X上安装MongoDB

在Mac OS X上安装MongoDB有三种方式，你可以直接从mongodb.org网站下载预编译的二进制包、使用包管理器，或者从源代码手工编译。随后的小节中我们会讨论前两种方式，随后再提供一些编译相关的说明。

1. 预编译的二进制包

先访问http://www.mongodb.org/downloads，你能看到所有可供下载的最新MongoDB二进制包，选择适用于你系统的最新稳定版本的下载URL。以下示例使用针对64位系统编译的MongoDB v2.0。

使用浏览器或`curl`工具下载压缩包，随后通过`tar`解压：

```
$ curl http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.0.0.tgz >
    mongo.tgz
$ tar xzf mongo.tgz
```

要运行MongoDB，你需要一个数据目录。`mongod`守护进程默认会把它的数据文件保存在

/data/db之中。创建该目录：

```
$ mkdir -p /data/db/
```

你已经可以启动服务器了，切换到MongoDB的bin目录，运行mongod可执行文件：

```
$ cd mongodb-osx-x86_64-2.0.0/bin
$ ./mongod
```

要是一切顺利，你应该可以看到一些和以下启动日志（经删减）类似的东西。请注意最后几行，确认服务器正监听默认的27017端口：

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
  pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

如果服务器意外终止，请阅读A.5节。

2. 使用包管理器

包管理器能极大简化MongoDB的安装，唯一的主要劣势是包维护方可能无法始终跟上最新的MongoDB版本。运行最新的稳定版本是很重要的，因此如果选择使用包管理器进行安装，请确保正在安装的MongoDB是最近的版本。

已知有两个Mac OS X的包管理器在维护最新版本的MongoDB，分别是MacPorts（http://www.macports.org）和Homebrew（http://mxcl.github.com/homebrew/）。要通过MacPorts进行安装，运行如下命令：

```
sudo port install mongodb
```

请注意，MacPorts会从头开始构建MongoDB及其所有依赖，如果选择这种方式，要做好心理准备，因为编译过程会很长。

与编译方式不同，Homebrew仅仅是下载最新的二进制包，因此速度会比MacPorts快很多。可以通过以下命令从Homebrew安装MongoDB：

```
$ brew update
$ brew install mongodb
```

安装完成后，Homebrew会提供一份指南，告诉你如何使用Mac OS X的Launch Agent来启动MongoDB。

A.1.3　在Windows上安装MongoDB

在Windows上安装MongoDB有两种方式。比较简单的一种，也是推荐的方式，是直接从mongodb.org网站上下载预编译的二进制包。你也可以从源代码进行编译，但这种做法只推荐给开发者和高级用户。下一节里可以看到与源代码编译相关的内容。

预编译二进制包

先访问http://www.mongodb.org/downloads，你能看到所有可供下载的最新MongoDB二进制

包，选择适用于你系统的最新稳定版本的下载URL。这里我们将安装针对64位Windows编译的MongoDB v2.0.

下载合适的发行包，然后进行解压。可以在Windows Explorer里找到MongoDB的.zip文件，右击后选择Extract All…，然后就能选择用于解压内容的文件夹了。

此外，也可以使用命令行。先进入Downloads目录，使用`unzip`工具来进行解压：

```
C:\> cd \Users\kyle\Downloads
C:\> unzip mongodb-win32-x86_64-2.0.0.zip
```

想运行MongoDB，你需要一个数据目录。`mongod`守护进程默认会把它的数据文件保存在C:\data\db之中。打开Windows的命令提示符，用如下方式创建文件夹：

```
C:\> mkdir \data
C:\> mkdir \data\db
```

你已经可以启动服务器了，切换到MongoDB的bin目录，运行`mongod`可执行文件：

```
C:\> cd \Users\kyle\Downloads
C:\Users\kyle\Downloads> cd mongodb-win32-x86_64-2.0.0\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin> mongod.exe
```

要是一切顺利，你应该可以看到一些和以下启动日志（经删减）类似的东西。请注意最后几行，确认服务器正监听默认的27017端口：

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
    pid=1773 port=27017 dbpath=/data/db/ 64-bit
Thu Mar 10 11:28:51 [initandlisten] db version v2.0.0, pdfile version 4.5
...
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
Thu Mar 10 11:28:51 [websvr] web admin interface listening on port 28017
```

如果服务器意外终止，请阅读A.5节。

最后，你要启动MongoDB Shell。为此，打开第二个终端窗口，然后执行mongo.exe。

```
C:\> cd \Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.0.0\bin> mongo.exe
```

A.1.4　从源代码进行编译

从源代码编译MongoDB，只推荐给高级用户和开发者采用。如果你只是想用用最新的东西，不用自己编译，可以从mongodb.org网站上下载最新修改的每日构建二进制包。

你可能想要自己编译MongoDB，其中最麻烦的部分是管理众多依赖，包括Boost、SpiderMonkey和PCRE。可以在http://www.mongodb.org/display/DOCS/Building找到各个平台的最新编译指南。

A.1.5　问题处理

虽然MongoDB易于安装，但用户偶尔还是会遇到一些小问题，通常表现为启动`mongod`守护进程时出现的错误消息。这里我会提供一份常见错误消息和解决方法的列表。

1. 错误的架构

如果试着在32位的机器上运行针对64位系统编译的二进制包，会看到如下错误：

```
-bash: ./mongod: cannot execute binary file
```

Windows 7上的消息更有帮助一些：

```
This version of
C:\Users\kyle\Downloads\mongodb-win32-x86_64-1.7.4\bin\mongod.exe
is not compatible with the version of Windows you're running.
Check your computer's system information to see whether you need
a x86 (32-bit) or x64 (64-bit) version of the program, and then
contact the software publisher.
```

解决方法是下载并运行32位的二进制包，在MongoDB的下载站点（http:// www.mongodb.org/downloads）上能找到适用于这两种架构的二进制包。

2. 数据目录不存在

MongoDB要有一个用于存放数据文件的目录。如果该目录不存在，你会看到如下错误：

```
dbpath (/data/db/) does not exist, terminating
```

解决方法就是创建该目录，查看前文的指南就能知道在你的操作系统上该怎么做了。

3. 权限不足

如果正运行在Unix类的系统上，需要确保运行mongod可执行文件的用户有写数据目录的权限。不然就会看到这样的错误：

```
Permission denied: "/data/db/mongod.lock", terminating
```

也可能是这样的错误：

```
Unable to acquire lock for lockfilepath: /data/db/mongod.lock, terminating
```

这两种情况下，都可以通过chmod或chown开启数据目录的权限，以此解决问题。

4. 无法绑定端口

MongoDB默认运行在27017端口，如果其他进程或是另一个mongod绑定了该端口，那么你就能看到这个错误：

```
listen(): bind() failed errno:98
    Address already in use for socket: 0.0.0.0:27017
```

有两种可行的解决方法。第一，找到运行于27017端口的其他进程，将其终止。此外，可以通过--port标志让mongod运行在另一个端口上。以下展示如何让MongoDB运行在27018端口：

```
mongod --port 27018
```

A.2　基本配置选项

本节我会简单介绍运行MongoDB时几个最常用的标志。

- **--dbpath**　指向存放数据文件的目录路径，默认是/data/db。
- **--logpath**　指向日志输出文件的路径。日志默认会输出在标准输出（stdout）里。

- **--port**　MongoDB监听的端口；如果没有指定，则设置为27017。
- **--rest**　该标志将开启简单REST接口，增强服务器的默认Web控制台。Web控制台总是运行在服务器监听端口之上的第1000个端口。因此，如果服务器在监听`localhost`的27017端口，那么Web控制台就在http://localhost:28017/。请花些时间研究Web控制台及其发布的命令，因为你能从中发现不少线上MongoDB服务器的信息。
- **--fork**　让进程以守护进程方式运行。请注意，`fork`只能用在Unix类的系统上。需要类似功能的Windows用户请查看指南，了解如何以Windows服务的方式运行MongoDB。可以在mongodb.org找到这些指南。

那些都是最重要的MongoDB启动标志，以下是在命令行中使用它们的例子：

```
$ mongod --dbpath /var/local/mongodb --logpath /var/log/mongodb.log
--port 27018 --rest --fork
```

请注意，你也可以在一个配置文件中指定全部这些选项。创建一个新的文本文件（称为mongodb.conf），内容如下：

```
dbpath=/var/local/mongodb
logpath=/var/log/mongodb.log
port=27018
rest=true
fork=true
```

在调用`mongod`时，通过`-f`选项来使用配置文件：

```
$ mongod -f mongodb.conf
```

如果连接上了一个MongoDB服务器，想知道启动时用了哪些选项，可以运行`getCmdLineOpts`命令获得一份启动选项列表：

```
> use admin
> db.runCommand({getCmdLineOpts: 1})
```

A.3　安装Ruby

本书中不少例子都是用Ruby编写的，为了能运行它们，你需要安装Ruby环境。也就是说要安装Ruby解释器和Ruby包管理器——RubyGems。

你应该使用版本号大于等于1.8.7的Ruby。在本书编写时，1.8.7和1.9.3是最常用的生产环境版本。

A.3.1　Linux与Mac OS X

Mac OS X和一些Linux发行版上默认装有Ruby。可以运行如下命令检查自己是否装有最近版本的Ruby解释器：

```
ruby -v
```

如果找不到该命令，或者运行的版本低于1.8.7，那你就该进行安装或升级了。在http://www.

ruby-lang.org/en/downloads/能找到详细的安装指南，帮助你在Mac OS X和一些Unix类的系统上安装Ruby。大多数包管理器（比如MacPorts和Aptitude）也维护了最近版本的Ruby，它们可能是获得一个可用Ruby环境的最简单途径。

除了Ruby解释器，你还需要Ruby包管理器RubyGems，用它来安装MongoDB的Ruby驱动。通过`gem`命令来确认是否安装了RubyGems：

```
gem -v
```

可以通过包管理器来安装RubyGems，但大多数用户会下载最新的版本，并且使用其中的安装程序。在https://rubygems.org/pages/download可以找到安装指南。

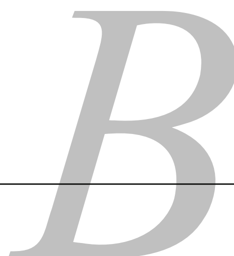
A.3.2　Windows

到目前为止，在Windows上安装Ruby和RubyGems最简单的途径是使用Windows Ruby Installer。可以在http://rubyinstaller.org/downloads找到安装程序。当你运行下载的可执行文件时，安装向导会指导你安装Ruby和RubyGems。

除了要安装Ruby，你还要安装Ruby DevKit，它能方便地编译Ruby的C扩展。MongoDB Ruby驱动的BSON库可能会使用这些扩展。

附　录

设计模式



B.1　模式

虽然不明显，但本书前面几章里有倡导大家使用一些设计模式。本附录中，我将总结那些模式，再补充一些没有提到的模式。

B.1.1　内嵌与引用

假设你在构建一个简单的应用程序，用MongoDB保存博客的文章和评论。该如何表示这些数据？在相应博客文章的文档里内嵌评论？还是说创建两个集合，一个保存文章，另一个保存评论，通过对象ID引用来关联评论和文章，这样会更好？

这里的问题是使用内嵌文档还是引用，这常常会给MongoDB的新用户带来困扰。幸好有些简单的经验法则，适用于大多数Schema设计场景：当子对象总是出现在父对象的上下文中时，使用内嵌文档；否则，将子对象保存在单独的集合里。

这对博客的文章和评论而言意味着什么？结论取决于应用程序。如果评论总是出现在博客的文章里，并且无需按照各种方式（根据发表日期、评论评价等）进行排序，那么内嵌的方式会更好。但是，如果说希望能够显示最新的评论，不管当前显示的是哪篇文章，那么就该使用引用。内嵌的方式可能性能稍好，但引用的方式更加灵活。

B.1.2　一对多

正如上一节所说的，可以通过内嵌或引用来表示一对多关系。当多端对象本质上属于它的父对象且很少修改时，应该使用内嵌。举个指南类应用程序（how-to application）的Schema作为例子，它能很好地说明这点。每个指南中的步骤都能表示为子文档数组，因为这些步骤是指南的固有部分，很少修改：

```
{ title: "How to soft-boil an egg",
          steps: [
          { desc: "Bring a pot of water to boil.",
            materials: ["water", "eggs"] },
          { desc: "Gently add the eggs a cook for four minutes.",
```

```
            materials: ["egg timer"]},
          { desc: "Cool the eggs under running water." },
        ]
}
```

如果两个相关条目要独立出现在应用程序里，那你就会想进行关联了。很多MongoDB的文章都建议在博客的文章里内嵌评论，认为这是一个好主意，但是关联会更灵活。如此一来，你可以方便地向用户显示他们的所有评论，还可以显示所有文章里的最新评论。这些特性对于大多数站点而言是必不可少的，但此时此刻却无法用内嵌文档来实现。[①]通常都会使用对象ID来关联文档，以下是一个示例文章对象：

```
{ _id: ObjectId("4d650d4cf32639266022018d"),
  title: "Cultivating herbs",
  text: "Herbs require occasional watering..."
}
```

下面是评论，通过`post_id`字段进行关联：

```
{ _id: ObjectId("4d650d4cf32639266022ac01"),
  post_id: ObjectId("4d650d4cf32639266022018d"),
  username: "zjones",
  text: "Indeed, basil is a hearty herb!"
}
```

文章和评论都放在各自的集合里，需要用两个查询来显示文章及其评论。因为会基于`post_id`字段查询评论，所以你希望为其添加一个索引：

```
db.comments.ensureIndex({post_id: 1})
```

我们在第4章、第5章和第6章中广泛使用了一对多模式，其中有更多例子可供参考。

B.1.3　多对多

在RDBMS里会使用联结表来表示多对多关系；在MongoDB里，则是使用数组键（array key）。本书先前的内容里就有该技术的示例，其中对产品和分类进行了关联。每个产品都包含一个分类ID的数组，产品与分类都有自己的集合。假设你有两个简单的分类文档：

```
{ _id: ObjectId("4d6574baa6b804ea563c132a"),
  title: "Epiphytes"
}
{ _id: ObjectId("4d6574baa6b804ea563c459d"),
  title: "Greenhouse flowers"
}
```

同时属于这两个分类的文档看起来会像下面这样：

```
{ _id: ObjectId("4d6574baa6b804ea563ca982"),
  name: "Dragon Orchid",
  category_ids: [ ObjectId("4d6574baa6b804ea563c132a"),
                  ObjectId("4d6574baa6b804ea563c459d") ]
}
```

① 有一个很热门的虚拟集合（virtual collection）特性请求，对两者都有很好的支持。请访问http://jira.mongodb.org/browse/SERVER-142了解这一特性的最新进展。

为了提高查询效率，应该为分类ID增加索引：

```
db.products.ensureIndex({category_ids: 1})
```

之后，查找Epiphytes分类里的所有产品，就是简单地匹配`category_id`字段：

```
db.products.find({category_id: ObjectId("4d6574baa6b804ea563c132a")})
```

要返回所有与Dragon Orchid产品相关的分类文档，先获取该产品的分类ID列表：

```
product = db.products.findOne({_id: ObjectId("4d6574baa6b804ea563ca982")})
```

然后使用`$in`操作符查询`categories`集合：

```
db.categories.find({_id: {$in: product['category_ids']}})
```

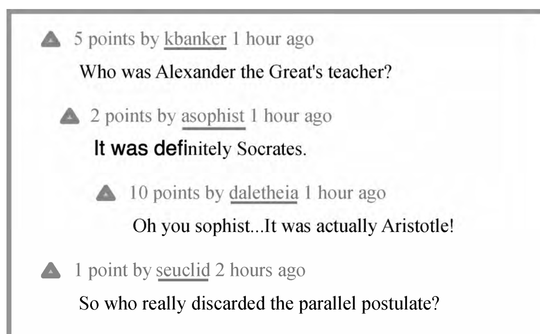
你会注意到，查询分类要求两次查询，而查询产品只需要一次。这是针对常见场景的优化，因为比起其他场景，查询某个分类里的产品可能性更大。

B.1.4　树

和大多数RDBMS一样，MongoDB没有内置表示和遍历树的机制。因此，如果你需要树的行为，就只有自己想办法了。我在第5章和第6章里给出了一种分类层级问题的解决方案，该策略是在每个分类文档里保存一份分类祖先的快照。这种去正规化让更新操作变复杂了，但是极大地简化了读操作。

可惜，去正规化祖先的方式并非适用于所有问题。另一个场景是在线论坛，成百上千的帖子通常层层嵌套，层次很深。对于祖先方式而言，这里的嵌套实在太多了，数据也太多了。有一个不错的解决方法——具化路径（materialized path）。

根据具化路径模式，树中的每个节点都要包含一个`path`字段，该字段具体保存了每个节点祖先的ID，根级节点有一个空`path`，因为它们没有祖先。让我们通过一个例子进一步了解该模式。首先，看看图B-1中的论坛帖子，其中是关于希腊历史的问题与回答。



图B-1　论坛里的帖子

让我们看看这些帖子是如何通过具化路径组织起来的。首先看到的是根级文档，所以`path`是`null`：

```
{ _id: ObjectId("4d692b5d59e212384d95001"),
  depth: 0,
  path: null,
  created: ISODate("2011-02-26T17:18:01.251Z"),
  username: "kbanker",
  body: "Who was Alexander the Great's teacher?",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

其他的根级文档，即用户seuclid提的问题，也有相同的结构。更能说明问题的是后续与亚历山大大帝（Alexander the Great）的老师相关的讨论。查看其中的第一个文档，我们注意到path中包含上级父文档的_id：

```
{ _id: ObjectId("4d692b5d59e212384d951002"),
  depth: 1,
  path: "4d692b5d59e212384d95001",
  created: ISODate("2011-02-26T17:21:01.251Z"),
  username: "asophist",
  body: "It was definitely Socrates.",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

下一个更深的文档里，path包含了根级文档和上级父文档的ID，依次用分号分隔：

```
{ _id: ObjectId("4d692b5d59e212384d95003"),
  depth: 2,
  path: "4d692b5d59e212384d95001:4d692b5d59e212384d951002",
  created: ISODate("2011-02-26T17:21:01.251Z"),
  username: "daletheia",
  body: "Oh you sophist...It was actually Aristotle!",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

最起码，你希望thread_id和path字段能加上索引，因为总是会基于其中某一个字段进行查询：

```
db.comments.ensureIndex({thread_id: 1})
db.comments.ensureIndex({path: 1})
```

现在的问题是如何查询并显示树。具化路径模式的好处之一是无论是要展现完整的帖子，还是其中的一棵子树，都只需查询一次数据库。前者的查询很简单：

```
db.comments.find({thread_id: ObjectId("4d692b5d59e212384d95223a")})
```

针对特定子树的查询稍微复杂一点，因为其中用到了前缀查询：

```
db.comments.find({path: /^4d692b5d59e212384d95001/})
```

该查询会返回拥有指定字符串开头路径的所有帖子。该字符串表示了用户名为kbanker的讨论的_id，如果查看每个子项的path字段，很容易发现它们都满足该查询。这种查询执行速度很快，因为这些前缀查询都能利用path上的索引。

获得帖子列表是很容易的事，因为它只需要一次数据库查询。但是显示就有点麻烦了，因为显示的列表中要保留帖子的顺序，这要在客户端做些处理——可以用以下Ruby方法实现。[①]第一

① 本书的源代码中包含了完整示例，其中实现了具化路径模式，并且用到了此处的显示方法。

个方法threaded_list构建了所有根级帖子的列表，还有一个Map，将父ID映射到子节点：

```
def threaded_list(cursor, opts={})
  list      = []
  child_map = {}
  start_depth = opts[:start_depth] || 0

  cursor.each do |comment|
    if comment['depth'] == start_depth
      list.push(comment)
    else
      matches = comment['path'].match(/([d|w]+)$/)
      immediate_parent_id = matches[1]
      if immediate_parent_id
        child_map[immediate_parent_id] ||= []
        child_map[immediate_parent_id] << comment
      end
    end
  end

  assemble(list, child_map)
end
```

assemble方法接受根节点列表和子节点Map，按照显示顺序构建一个新的列表：

```
def assemble(comments, map)
  list = []
  comments.each do |comment|
    list.push(comment)
    child_comments = map[comment['_id'].to_s]
    if child_comments
      list.concat(assemble(child_comments, map))
    end
  end

  list
end
```

到了真正显示的时候，只需迭代这个列表，根据每个讨论的深度适当缩进就行了：

```
def print_threaded_list(cursor, opts={})
  threaded_list(cursor, opts).each do |item|
    indent = "  " * item['depth']
    puts indent + item['body'] + " #{item['path']}"
  end
end
```

此时，查询并显示讨论的代码就很简单了：

```
cursor = @comments.find.sort("created")
print_threaded_list(cursor)
```

B.1.5 工作队列

你可以使用标准集合或者固定集合在MongoDB里实现工作队列。无论使用哪种集合，findAndModify命令都能让你原子地处理队列项。

队列项要求有一个状态字段（state）和一个时间戳字段（timestamp），剩下的字段用来包含其承载的内容。状态可以编码为字符串，但是整数更省空间。我们将用0和1来分别表示未处理和已处理。时间戳是标准的BSON日期。此处承载的内容就是一个简单的纯文本消息，它原则上可以是任何东西。

```
{ state: 0,
  created: ISODate("2011-02-24T16:29:36.697Z"),
  message: "hello world" }
```

你需要声明一个索引，这样才能高效地获取最老的未处理项（FIFO）。state和created上的复合索引正好合适：

```
db.queue.ensureIndex({state: 1, created: 1})
```

随后使用findAndModify返回下一项，并将其标记为已处理：

```
q = {state: 0}
s = {created: 1}
u = {$set: {state: 1}}
db.queue.findAndModify({query: q, sort: s, update: u})
```

如果使用的是标准集合，需要确保会删除老的队列项。可以在处理时使用findAndModify的{remove: true}选项来移除它们。但是有些应用程序希望处理完成之后，过一段时间再进行删除操作。

固定集合也能作为工作队列的基础。没有_id上的默认索引，固定集合在插入时速度更快，但是这一差别对于大多数应用程序而言都可以忽略不计。另一个潜在的优势是自动删除特性，但这一特性是一把双刃剑：你要确保集合足够大，避免未处理的队列项被挤出队列。因此，如果使用固定集合，要让它足够大，理想的集合大小取决于队列的写吞吐量和平均载荷内容大小。

一旦决定了固定集合的大小，Schema、索引和findAndModify的使用都和刚才介绍的标准集合一样。

B.1.6　动态属性

MongoDB的文档数据模型在表示属性会有变化的条目时非常有用。产品就是一个公认的例子，在本书先前的部分里你已经看到过此类建模方法了。将此类属性置于子文档之中，就是一种行之有效的建模方法。在一个products集合中，可以保存完全不同的产品类型，你可以保存一副耳机：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ebd-123"
  name: "Hi-Fi Earbuds",
  type: "Headphone",
  attrs: { color: "silver",
           freq_low: 20,
           freq_hi: 22000,
           weight: 0.5
         }
}
```

和一块SSD硬盘：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ssd-456"
  name: "Mini SSD Drive",
  type: "Hard Drive",
  attrs: { interface: "SATA",
           capacity: 1.2 * 1024 * 1024 * 1024,
           rotation: 7200,
           form_factor: 2.5
         }
}
```

如果需要频繁地查询这些属性，可以为它们建立稀疏索引。例如，可以为常用的耳机范围查询进行优化：

```
db.products.ensureIndex({"attrs.freq_low": 1, "attrs.freq_hi": 1},
  {sparse: true})
```

还可以通过以下索引，根据转速高效地查询硬盘：

```
db.products.ensureIndex({"attrs.rotation": 1}, {sparse: true})
```

此处的整体策略是为了提高可读性和应用可发现性（discoverability）而将属性圈在一个范围里，通过稀疏索引将空值排除在索引之外。

如果属性是完全不可预测的，那就无法为每个属性构建单独的索引。这就必须使用不同的策略了，就像下面这个示例文档所示：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ebd-123"
  name: "Hi-Fi Earbuds",
  type: "Headphone",
  attrs: [ {n: "color", v: "silver"},
           {n: "freq_low", v: 20},
           {n: "freq_hi", v: 22000},
           {n: "weight", v: 0.5}
         ]
}
```

这里的`attrs`指向一个子文档数组，每个子文档都有两个值`n`和`v`，分别对应了动态属性的名字和取值。这种正规化表述让你能通过一个复合索引来索引这些属性：

```
db.products.ensureIndex({"attrs.n": 1, "attrs.v": 1})
```

随后就能用这些属性进行查询了，但是必须使用`$elemMatch`查询操作符：

```
db.products.find({attrs: {$elemMatch: {n: "color", v: "silver"}}})
```

请注意，这种策略会带来不少开销，因为它要在索引里保存键名。在用于生产环境之前，使用有代表性的数据集进行性能测试是很重要的。

B.1.7　事务

MongoDB不会为一系列操作提供ACID保障，也不存在与RDBMS里的`BEGIN`、`COMMIT`和

ROLLBACK语义等价的东西。需要这些特性时，就换个数据库吧（可以针对需要适当事务保障的数据部分，也可以把应用程序的数据库整个换了）。不过MongoDB支持单个文档的原子性、持久化更新，还有一致性读，这些特性虽然原始，但能在应用程序里实现类似事务的作用。

第6章在处理订单授权与库存管理时已经有一个很好的例子了。本附录前面实现的工作队列也能方便地添加回滚支持。这两个例子里，功能强大的findAndModify命令是实现类似事务行为的基础，可以用来操作一个或多个文档的state字段。

所有这些案例里用到的事务策略都能描述为补偿驱动（compensation-driven）[①]。抽象后的补偿过程如下。

(1) 原子性地修改文档状态。

(2) 执行一些操作，可能包含对其他文档的原子性修改。

(3) 确保整个系统（所有涉及的文档）都处于有效状态。如果情况如此，标记事务完成；否则将每个文档都改回事务前的状态。

值得注意的是，补偿驱动策略几乎是长时间多步事务所必不可少的，授权、送货及取消订单的过程只是一个例子。对于这些场景，就算是有完整事务语义的RDBMS也必须实现一套类似的策略。

也许没办法避开某些应用程序对多对象ACID事务的需求。但是只要有正确的模式，MongoDB也能提供一些事务保障，可以支持应用程序所需的事务性语义。

B.1.8　局部性与预计算

MongoDB经常被冠以分析数据库（analytics database）之名，大量用户在MongoDB之中保存分析数据。原子增加与富文档的结合看上去很棒。例如，下面这个文档表示了一个月中每一天的总页面访问量，还带有该月的总访问量。简单起见，以下文档只包含该月头五天的数据：

```
{ base: "org.mongodb", path: "/",
  total: 99234,
  days: {
    "1": 4500,
    "2": 4324,
    "3": 2700,
    "4": 2300,
    "5": 0
  }
}
```

可以使用$inc操作符进行简单的针对性更新，以修改某一天或这个月的访问量：

```
use stats-2011
db.sites-nov.update({ base: "org.mongodb", path: "/" },
  $inc: {total: 1, "days.5": 1 });
```

① 有两个涉及补偿驱动事务的文献值得一读。最初由Garcia-Molina和Salem所著的“Sagas”（http://mng.bz/73is）。另一篇不太正式，但同样有趣，见“Your Coffee Shop Doesn’t Use Two-Phase Commit”（http://mng.bz/kpAq），作者是Gregor Hohpe。

稍微关注一下集合与数据库的名字，集合`sites-nov`是针对某一月份的，而数据库`stats-2011`是针对特定年份的。

这为应用程序带来了良好的局部性。在查询最近的访问情况时，只需要查询一个集合，比起整个分析历史数据，这数量就小多了。如果需要删除数据，可以删掉某个时间段的集合，而不是从较大的集合里删除文档的子集。后者通常会造成磁盘碎片。

实践中的另一条原则是预计算。有时，在每个月开头时，你需要插入一个模板文档，其中每一天都是零值。因此，在增加计数器时文档大小不会改变，因为并没有增加字段，只是原地改变了它们的值。这一点很重要，因为在写操作时，这能避免对文档重新进行磁盘分配。重新分配很慢，通常也会造成碎片。

B.2　反模式

MongoDB缺乏约束，这会导致糟糕的数据组织。在一些有问题的生产环境中，经常会出现如下情况。

B.2.1　索引随意

当用户遭遇性能问题时，经常会发现一大堆无用的或是低效的索引。对于应用程序而言，最有效的索引集总是基于对其运行的查询的分析。请遵循第7章里的优化方法。

B.2.2　类型杂乱

请确保一个集合里的同名键都拥有一样的类型。举个例子，如果要保存一个电话号码，就用一致的方式保存，可以是字符串，也可以是整数（但别一起用）。在某个键值里混用不同类型会造成应用程序逻辑复杂，在某些强类型语言中这会让BSON文档难以解析。

B.2.3　桶集合

一个集合应该只针对一类实体，不要把产品和用户放在一个集合里。因为集合的代价并不高，所以应用程序里的每个类型都应该有自己的集合。

B.2.4　文档很大、嵌套很深

关于MongoDB的文档数据模型有两个误解。其一，永远不要在集合之间构建关系，而是在一个文档里表示所有的关系。这通常会演变为一片混乱，但是用户有时还是乐此不疲。第二个误解源自对“文档”这个词的过度字面解释，在用户看来，文档是一个实体，就像真实生活中的文档。这会导致文档过大，不易查询和更新，理解就更谈不上了。

这里的底线是应该保持小文档（每个文档最好都能小于100 KB，除非是在保存原始二进制数据），内嵌层次不易过深。文档尺寸较小会让更新的开销更少，当需要在磁盘上完整重写一遍文

档时，重写的东西会更少。另一个好处是文档依然可以理解，这能让需要理解数据模型的开发者生活更轻松。

B.2.5　一个用户一集合

为每个用户构建一个集合通常都不是好主意。这种做法的问题之一是命名空间（索引加集合）会超过默认的24 000。一旦超过这个阈值，就必须分配新的数据库。此外，每个集合和它的索引都会引入额外的开销，因此这种策略很浪费空间。

B.2.6　无法分片的集合

如果预计到一个集合会变大，可能大到需要分片，这时需要确保最终能进行分片。只有那些能定义出高效分片键的集合，才是可分片的。请回顾第9章关于分片键选取的内容。

附录C 二进制数据与GridFS

在存储图片、缩略图、音频和其他二进制文件时，很多应用程序都只依赖文件系统。虽然文件系统提供了对文件的快速访问能力，但也会带来组织混乱问题。考虑到大多数文件都限制了每个目录的文件数，如果要保存数以百万的文件，需要设计一套策略，将文件放入多个目录里。另一个难点涉及元数据，因为文件元数据仍然存储在数据库里，所以想对文件及其元数据进行精确备份会极其复杂。

针对某些使用场景，直接将文件保存在数据库里更加合理，因为这能简化文件的组织与备份。在MongoDB中，可以使用BSON二进制类型来保存各种二进制数据。这种数据类型与RDBMS BLOB（binary large object）类型相对应，是MongoDB提供的两种二进制对象存储方式的基础。

第一种方式，每个文件一个文档，适用于较小的二进制对象。如果要对大量缩略图或MD5进行分类保存，那么单一文档二进制存储会更简单一些。另一方面，你可能希望保存大图片和音频文件。这时GridFS会是更好的选择，它是MongoDB用于存储任意大小二进制对象的API。下文中你会看到这两种存储技术的复杂示例。

C.1 简单二进制存储

BSON中包含了一种很适用于二进制数据的类型，可以用它直接在MongoDB文档中保存二进制对象。对象大小的唯一限制是文档本身的大小限制，MongoDB v2.0起是16 MB。因为这样的大文档会消耗系统资源，所以在保存大于1 MB的二进制对象时鼓励使用GridFS。

我们会看到两种在单个文档里保存二进制对象的合理用法，先是保存图片缩略图，然后是保存附属的MD5。

C.1.1 保存缩略图

假设需要保存一组图片缩略图，代码很简单。首先，获得图片的文件名——`canyon-thumb.jpg`，将数据读取到局部变量中；然后，使用Ruby驱动的`BSON::Binary`构造器将原始的二进制数据封装为BSON二进制对象：

```
require 'rubygems'
require 'mongo'

image_filename = File.join(File.dirname(__FILE__), "canyon-thumb.jpg")
image_data = File.open(image_filename).read

bson_image_data = BSON::Binary.new(image_data)
```

剩下的就是构建一个将包含二进制数据的简单文档，随后将其插入数据库：

```
doc = {"name" => "monument-thumb.jpg",
       "data" => bson_image_data }

@con = Mongo::Connection.new
@thumbnails = @con['images']['thumbnails']
@image_id = @thumbnails.insert(doc)
```

要提取二进制数据，先获取文档，在Ruby里，`to_s`方法会将数据拆解（unpack）成二进制字符串，你可以用它来对保存的数据和原始数据进行比较：

```
doc = @thumbnails.find_one({"_id" => @image_id})
if image_data == doc["data"].to_s
  puts "Stored image is equal to the original file!"
end
```

如果运行上述脚本，你会看到一段消息，这个消息表明两个文件是相同的。

C.1.2　存储MD5

将校验和（checksum）存储为二进制数据是很常见的做法，这是BSON二进制类型的另一种用途。以下展示如何生成缩略图的MD5，并将它添加到刚才保存的文档里：

```
require 'md5'
md5 = Digest::MD5.file(image_filename).digest
bson_md5 = BSON::Binary.new(md5, BSON::Binary::SUBTYPE_MD5)

@thumbnails.update({:_id => @image_id}, {"$set" => {:md5 => bson_md5}})
```

请注意，在创建BSON二进制对象时，要将数据标记为`SUBTYPE_MD5`。子类型是BSON二进制类型的额外字段，标明了所保存的二进制数据种类。但是，该字段是完全可选的，对数据库如何保存或解释数据没有影响。①

查询刚才保存的文档非常容易，但请注意，你应该排除数据字段，这样返回的文档尺寸较小且方便阅读：

```
> use images
> db.thumbnails.findOne({}, {data: 0})
{
  "_id" : ObjectId("4d608614238d3b4ade000001"),
  "md5" : BinData(5,"K1ud3EUjT49wdMdkOGjbDg=="),
  "name" : "monument-thumb.jpg"
}
```

① 事实并非总是如此。已经不再推荐使用默认子类型2了，该类型指出在附带的二进制数据中还包含额外的四个字节，它们用来标明大小，这确实对一些数据库命令有影响。现在的默认子类型是0，目前所有的子类型都以相同的方式来保存二进制负载内容。因此也可以将子类型视为一种轻量级的标签，应用程序开发者可以选择使用。

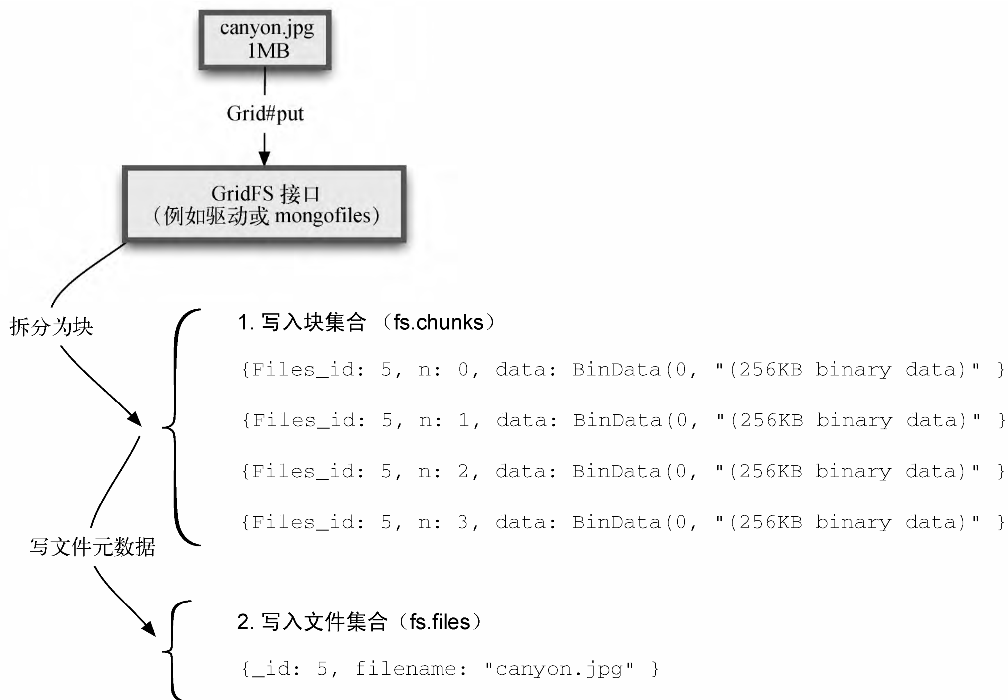
md5字段被清晰地标记为二进制数据，带有子类型和原始的负载内容。

C.2　GridFS

在MongoDB中，常见做法是使用GridFS保存任意大小的文件。所有的官方驱动和MongoDB的mongofiles工具都实现了GridFS规范，这保证了跨平台的一致性访问能力。GridFS对于在数据库中存储大型二进制对象很有用。通常GridFS也能快速处理这些对象，而且存储方法有助于用流进行操作。

GridFS这个词通常会带来一些困扰，所以最好立刻澄清两件事情。第一，GridFS并非MongoDB的内部特性，正如前面所说的，GridFS是所有官方驱动（和一些工具）用来在数据库里管理大型二进制对象的一个惯例。第二，必须说明GridFS没有真实文件系统那样丰富的语义。举例来说，GridFS中没有锁和并发的协议，这就把GridFS接口限制在了简单的提交（put）、获取（get）和删除（delete）操作上。也就是说，如果想更新一个文件，需要先删除，然后再提交新版本。

GridFS会把大文件拆分成小的256 KB的块，将每个块都保存在单独的文档里。这些块默认保存在名为fs.chunks的集合里。写完块后，文件的元数据会被放到名为fs.files的另一集合里，用单独的文档来保存。图C-1简单描述了这个流程，处理一个假想的1 MB文件——canyon.jpg。



图C-1　使用GridFS保存文件

要运用GridFS，有这些理论基础就够了。接下来，让我们通过Ruby的GridFS API和mongofiles工具来进行一些实践。

C.2.1　通过Ruby使用GridFS

先前你保存了一个小的缩略图，缩略图只有10 KB，可以理想地保存在单个文档里。原始图片有差不多2 MB，因此更适合用GridFS来存储，这里将通过Ruby的GridFS API来存储原始图片。首先，连接数据库，初始化一个Grid对象，其中会持有一个用来保存GridFS文件的数据库的引用。

接下来，打开原始图片canyon.jpg，准备读取文件内容。最基本的GridFS接口通过一些方法来提交和获取文件。你可以使用Grid#put方法，它既可以接受二进制数据字符串，也可以接受IO对象，比如文件指针。传入文件指针后，数据就写进数据库了。

该方法会返回文件的唯一对象ID：

```
@con  = Mongo::Connection.new
@db   = @con["images"]

@grid = Mongo::Grid.new(@db)

filename = File.join(File.dirname(__FILE__), "canyon.jpg")
file = File.open(filename, "r")

file_id = @grid.put(file, :filename => "canyon.jpg")
```

如上所述，GridFS用了两个集合来存储文件数据。第一个集合通常名为fs.files，用来保存每个文件的元数据。第二个集合fs.chunks保存了每个文件的一个或多个二进制数据块。让我们在Shell里简单实验一下。

切换到images数据库，查询fs.files集合里的第一个条目。你会看到刚才保存的文件的元数据：

```
> use images
> db.fs.files.findOne()
{
  "_id" : ObjectId("4d606588238d3b4471000001"),
  "filename" : "canyon.jpg",
  "contentType" : "binary/octet-stream",
  "length" : 2004828,
  "chunkSize" : 262144,
  "uploadDate" : ISODate("2011-02-20T00:51:21.191Z"),
  "md5" : "9725ad463b646ccbd287be87cb9b1f6e"
}
```

这些是每个GridFS文件最起码的属性，大多数的含义不言而喻。你可以看到这个文件大约有2 MB，被拆分成了256 KB的块。其中还有一个MD5，GridFS规范要求有一个校验和来保证存储的文件和原始文件是一致的。

每个块都在files_id字段里保存了文件的对象ID，可以方便地计算出该文件所使用的块的数量：

```
> db.fs.chunks.count({"files_id" : ObjectId("4d606588238d3b4471000001")})
8
```

有了块大小和文件总大小，应该能算出有八个块。你还可以方便地看到块本身的内容。与之前一样，要排除数据以保持输出易于阅读。以下查询会返回八个块中的第一个块，n的值标识了它的序号：

```
> db.fs.chunks.findOne({files_id: ObjectId("4d606588238d3b4471000001")},
          {data: 0})
{
  "_id" : ObjectId("4d606588238d3b4471000002"),
  "n" : 0,
  "files_id" : ObjectId("4d606588238d3b4471000001")
}
```

读取GridFS文件就和写入一样方便。在下面的例子里，使用Grid#get返回一个类似IO的GridIO对象，表示该文件。然后就能将GridFS文件用流的方式写回文件系统，这里每次读取256 KB写入原始文件的副本中：

```
image_io = @grid.get(file_id)

copy_filename = File.join(File.dirname(__FILE__), "canyon-copy.jpg")
copy = File.open(copy_filename, "w")

while !image_io.eof? do
  copy.write(image_io.read(256 * 1024))
end

copy.close
```

随后可以验证一下，两个文件是一样的：[①]

```
$ diff -s canyon.jpg canyon-copy.jpg
Files canyon.jpg and canyon-copy.jpg are identical
```

以上就是通过驱动读写GridFS文件的基本操作。不同的GridFS API之间稍有不同，但有了上面的例子和关于GridFS工作原理的基本知识，理解你所用驱动的文档应该不成问题。

C.2.2　通过mongofiles操作GridFS

MongoDB发行包里包含了一个非常实用的工具，名为mongofiles，它可以在命令行里罗列、提交、获取和删除GridFS文件。例如，你可以列出images数据库里的GridFS文件：

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg  2004828
```

你还可以方便地添加文件，下面演示如何添加刚才用Ruby脚本写入的图像文件副本：

```
$ mongofiles -d images put canyon-copy.jpg
connected to: 127.0.0.1
added file: { _id: ObjectId('4d61783326758d4e6727228f'),
              filename: "canyon-copy.jpg",
              chunkSize: 262144, uploadDate: new Date(1298233395296),
              md5: "9725ad463b646ccbd287be87cb9b1f6e", length: 2004828 }
```

① 这段代码假设你已经安装了diff工具。

可以再次查看文件列表，验证一下文件是否已写入：

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg  2004828
canyon-copy.jpg  2004828
```

mongofiles支持不少选项，而你可以通过--help参数查看这些选项：

```
$ mongofiles --help
```

附　录

在PHP、Java与C++中使用MongoDB

D

本书透过JavaScript和Ruby的视角来展示MongoDB，但是还有很多与MongoDB通信的其他方式，本附录就会展示其中的三种。我会先从PHP开始，因为它是流行的脚本语言。包含Java是因为它仍是企业开发领域的霸主，对本书的很多读者而言很重要。而且，Java驱动的API与大多数脚本语言驱动的API差别很大。最后，摆出C++驱动是因为它是MongoDB代码中的一块核心部分，对于那些想要构建高性能独立应用程序的开发者而言很可能非常有用。

每个语言的小节中，我都会描述如何构造文档、建立连接，最后演示一个完整的程序，它可以插入、修改、查询和删除示例文档。所有的程序都会执行相同的操作，产生相同的输出，因此很容易进行比较。每个程序里的文档都是一个简单Web爬虫要保存的示例文档；下面是用JSON表示的文档，仅供参考：

```
{ url: "org.mongodb",
  tags: ["database", "open-source"],
  attrs: { "last-visit" : ISODate("2011-02-22T05:18:28.740Z"),
           "pingtime" : 20
         }
}
```

D.1　PHP

PHP社区热情地接纳了MongoDB，并提供了高质量的驱动作为回馈。示例代码看上去和等价的Ruby代码基本上是相似的。

D.1.1　文档

PHP的数组是由有序字典（ordered dictionary）来实现的，因此能很好地映射到BSON文档上。可以使用PHP的`array`按照文档的字面内容创建一个简单的文档：

```
$basic = array( "username" => "jones", "zip" => 10011 );
```

PHP的数组也可以相互嵌套。以下这个复杂文档包含一个标签数组和一个子文档，子文档中带有日期`last_access`和整数`pingtime`。请注意，你必须使用特殊的`MongoDate`类来表示日期：

```
$doc = array( "url"    => "org.mongodb",
              "tags"   => array( "database", "open-source"),
              "attrs"  => array( "last_access" => new MongoDate(),
                                 "pingtime" => 20
                               )
            );
```

D.1.2　连接

可以通过Mongo构造器连接到单个节点：

```
$conn = new Mongo( "localhost", 27017 );
```

要连接副本集，就为Mongo构造器传入一个MongoDB连接URI，同时还要指定array("replicaSet" => true)：

```
$repl_conn = new Mongo( "mongo://localhost:30000,localhost:30001",
              array( "replicaSet" => true ));
```

MongoDB连接URI

MongoDB连接URI是在不同驱动间指定连接选项的标准方式。大多数驱动都会接受连接URI，这对跨环境访问MongoDB服务器的系统而言可以简化配置。请访问官方的线上MongoDB文档以了解最新的URI规范。

通常PHP应用程序使用持久化连接时性能会更好。如果要使用持久化连接，请确保添加了array("persistent" => "x")，其中"x"表示所创建的持久化连接的唯一标识符：

```
$conn = new Mongo( "localhost", 27017, array( "persist" => "x" ) );
```

D.1.3　示例程序

以下PHP程序演示了如何插入、更新、查询和删除一个文档，同时还包含了几个PHP BSON文档的表述。

代码清单D-1　PHP驱动的用法示例

```
<?php
  $m = new Mongo( "localhost", 27017 );
  $db = $m->crawler;
  $coll = $db->sites;

  $doc = array( "url"    => "org.mongodb",
                "tags"   => array( "database", "open-source"),
                "attrs"  => array( "last_access" => new MongoDate(),
                                   "pingtime" => 20
                                 )
              );

  $coll->insert( $doc );
```

```
  print "Initial document:n";
  print print_r( $doc );

  print "Updating pingtime...n";
  $coll->update(
    array( "_id"  => $doc["_id"] ),
    array( '$set' => array( 'attrs.pingtime' => 30 ) )
  );

  print "After update:n";
  $cursor = $coll->find();
  print print_r( $cursor->getNext() );

  print "nNumber of site documents: " . $coll->count() . "n";

  print "Removing documents...n";
  $coll->remove();
?>
```

D.2 Java

在众多MongoDB驱动之中，Java驱动也许是在生产环境里使用最频繁的一个。除了后台是纯Java的应用之外，Java驱动也是各种JVM语言驱动的基础，比如Scala、Clojure和JRuby。Java中缺乏按字面义表示的字典，这让BSON文档的构建略显复杂，但就整个驱动而言，使用起来还算方便。

D.2.1 文档

要构造BSON文档，可以初始化一个`BasicBSONObject`实例，它实现了`Map`接口，围绕`get()`和`put()`操作提供了一套简单的API。

方便起见，`BasicBSONObject`构造器接受一个可选的初始化键值对，用它就能构造一个简单的文档，比如像下面这样：

```
DBObject simple = new BasicDBObject( "username", "Jones" );
simple.put( "zip", 10011 );
```

添加子文档意味着创建一个额外的`BasicBSONObject`，其中的数组就是普通Java数组：

```
DBObject doc = new BasicDBObject();
String[] tags = { "database", "open-source" };

doc.put("url", "org.mongodb");
doc.put("tags", tags);

DBObject attrs = new BasicDBObject();
attrs.put( "lastAccess", new Date() );
attrs.put( "pingtime", 20 );

doc.put( "attrs", attrs );

System.out.println( doc.toString() );
```

最后请注意，你可以通过文档的`toString()`方法来查看它。

D.2.2　连接

创建一个单节点连接是件很容易的事情，只要记得把调用封装在一个try代码块里就行了：

```
try {
  Mongo conn = new Mongo("localhost", 27017);
} catch (Exception e) {
  throw new RuntimeException(e);
}
```

要连接副本集，先构造一个ServerAddress对象列表，将它传给Mongo构造器：

```
List servers = new ArrayList();
servers.add( new ServerAddress( "localhost" , 30000 ) );
servers.add( new ServerAddress( "localhost" , 30001 ) );

try {
  Mongo replConn = new Mongo( servers );
} catch (Exception e) {
  throw new RuntimeException(e);
}
```

Java驱动中为写关注提供了灵活的支持，可以在Mongo、DB、DBCollection对象以及DBCollection的任意写方法上指定不同的写关注。这里我们通过WriteConcern配置类在连接上指定了全局写关注：

```
WriteConcern w = new WriteConcern( 1, 2000 );
conn.setWriteConcern( w );
```

D.2.3　示例程序

下面这段Java程序直接翻译了先前的PHP程序，应该没什么看不懂的地方：

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.WriteConcern;
import java.util.Date;

public class Sample {

  public static void main(String[] args) {

    Mongo conn;
    try {
      conn = new Mongo("localhost", 27017);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }

    WriteConcern w = new WriteConcern( 1, 2000 );
    conn.setWriteConcern( w );

    DB db = conn.getDB( "crawler" );
    DBCollection coll = db.getCollection( "sites" );
```

```
        DBObject doc = new BasicDBObject();
        String[] tags = { "database", "open-source" };

        doc.put("url", "org.mongodb");
        doc.put("tags", tags);

        DBObject attrs = new BasicDBObject();
        attrs.put( "lastAccess", new Date() );
        attrs.put( "pingtime", 20 );

        doc.put( "attrs", attrs );

        coll.insert(doc);

        System.out.println( "Initial document:n" );
        System.out.println( doc.toString() );

        System.out.println( "Updating pingtime...n" );
        coll.update( new BasicDBObject( "_id", doc.get("_id") ),
           new BasicDBObject( "$set", new BasicDBObject( "pingtime", 30 ) ) );

        DBCursor cursor = coll.find();

        System.out.println( "After updaten" );
        System.out.println( cursor.next().toString() );

        System.out.println( "Number of site documents: " + coll.count() );

        System.out.println( "Removing documents...n" );
        coll.remove( new BasicDBObject() );
    }
}
```

D.3　C++

推荐C++驱动的原因有两点，其一是速度，其二是它与核心服务器关系密切。你渴望找到更快的驱动，而且如果对MongoDB的内部实现感兴趣，学习C++驱动是了解源代码的一个不错的切入点。C++驱动并非独立的驱动，而是作为构成内部MongoDB API所必须的代码混在核心代码之中。但是，也有办法将这些代码用作独立的库。

D.3.1　文档

在C++中有两种创建BSON文档的途径。可以使用有些冗长的`BSONObjBuilder`，也可以用封装了它的`BSON`宏。我会针对每个示例文档分别演示这两种方式。

让我们先从一个简单的文档入手：

```
BSONObjBuilder simple;
simple.genOID().append("username", "Jones").append( "zip", 10011 );
BSONObj doc = simple.obj();

cout << doc.jsonString();
```

请注意，你显式地使用`getOID()`函数生成了对象ID。C++的BSON对象是静态的，这意味着插入函数不能像在其他驱动中那样修改BSON对象。如果想要在插入后取得对象ID，你需要自己来生成它。

还要注意你必须在使用前将BSONObjBuilder转换为BSONObj。可以通过调用BSONObj-Builder的obj()方法进行转换。

现在，让我们使用辅助宏来生成相同的文档，BSON和GENOID能让你少打不少字：

```
BSONObj o = BSON( GENOID << "username" << "Jones" << "zip" << 10011 );
cout << o.jsonString();
```

构造更复杂的文档会让人想起Java，你必须分别构造每个子对象。请注意，你是通过标准的BSONObjBuilder来构建数组的，只是在其中使用了数字字符标志0和1。事实上，BSON中也是这样保存数组的：

```
BSONObjBuilder site;
site.genOID().append("url", "org.mongodb");
BSONObjBuilder tags;
tags.append("0", "database");
tags.append("1", "open-source");
site.appendArray( "tags", tags.obj() );

BSONObjBuilder attrs;
time_t now = time(0);
attrs.appendTimeT( "lastVisited", now );
attrs.append( "pingtime", 20 );
site.append( "attrs", attrs.obj() );

BSONObj site_obj = site.obj();
```

和之前一样，为了使代码更简洁，你更青睐于使用宏。要特别注意BSON_ARRAY和DATENOW这两个宏，关注它们替换的BSONObjBuilder版本所构造出的文档：

```
BSONObj site_concise = BSON( GENOID << "url" << "org.mongodb"
  << "tags" << BSON_ARRAY( "database" << "open-source" )
  << "attrs" << BSON( "lastVisited" << DATENOW << "pingtime" << 20 ) );
```

唯独C++里有一个要求：必须显式地标记用作查询选择器的BSON文档。一种实现方式是使用Query()构造器：

```
BSONObj selector = BSON( "_id" << 1 );
Query * q1 = new Query( selector );
cout << q1->toString() << "n";
```

同样的，方便的QUERY宏一般会更受青睐：

```
Query q2 = QUERY( "pingtime" << LT << 20 );
cout << q2.toString() << "n";
```

D.3.2　连接

可以方便地实例化一个DBClientConnection创建单节点连接，并且始终将代码封装在try代码块里：

```
DBClientConnection conn;

try {
  conn.connect("localhost:27017");
}
```

```
catch( DBException &e ) {
    cout << "caught " << e.what() << endl;
}
```

连接副本集，先要构建一个包含`HostAndPort`对象的`vector`，然后将副本集的名字以及`vector`一起传给`DBClientReplicaSet`构造器。可以调用`toString()`来检查对象的内容：

```
std::vector<HostAndPort> seeds (2);
seeds.push_back( HostAndPort( "localhost", 30000 ) );
seeds.push_back( HostAndPort( "localhost", 30001 ) );

DBClientReplicaSet repl_conn( "myset", seeds );
try {
  repl_conn.connect();
catch( DBException &e ) {
    cout << "caught " << e.what() << endl;
}

cout << repl_conn.toString();
```

D.3.3　示例程序

在C++代码示例中，主要注意到一点，没有明确的类来抽象数据库和集合。所有的插入、更新、查询和删除都直接通过连接对象本身来执行。你以命名空间的形式（`crawler.sites`）来指定数据库和集合，将它作为这些方法的第一个参数：

```
#include <iostream>
#include <ctime>
#include "client/dbclient.h"

using namespace mongo;

int main() {
    DBClientConnection conn;

    try {
      conn.connect("localhost:27017");
    }
    catch( DBException &e ) {
        cout << "caught " << e.what() << endl;
    }

    BSONObj doc = BSON( GENOID << "url" << "org.mongodb"
      << "tags" << BSON_ARRAY( "database" << "open-source" )
      << "attrs" << BSON( "lastVisited" << DATENOW << "pingtime" << 20 ) );

    cout << "Initial document:n" << doc.jsonString() << "n";
    conn.insert( "crawler.sites", doc );

    cout << "Updating pingtime...n";
    BSONObj update = BSON( "$set" << BSON( "attrs.pingtime" << 30) );
    conn.update( "crawler.sites", QUERY("_id" << doc["_id"]), update);

    cout << "After update:n";
    auto_ptr<DBClientCursor> cursor;
    cursor = conn.query( "crawler.sites", QUERY( "_id" << doc["_id"]) );
```

```
    cout << cursor->next().jsonString() << "n";

    cout << "Number of site documents: " <<
      conn.count( "crawler.sites" ) << "n";

    cout << "Removing documents...n";
    conn.remove( "crawler.sites", BSONObj() );

    return 0;
}
```

附　录

空间索引



随着智能移动设备的增长，对基于位置服务的需求正稳步提升。要构建这些与位置相关的应用程序，数据库需要能索引并查询空间数据。这些特性很早就加入了MongoDB的线路图，虽说MongoDB的空间索引（spatial indexing）还没有达到PostGIS这种完整的功能，但已经可以支撑许多流行站点的位置查询了。[①]

正如其名所示，空间索引针对表示位置的数据进行了优化。在MongoDB中，这类数据通常表示为地理坐标系中的经度和纬度，其上的空间索引允许基于用户的位置进行查询。例如，你有一个集合，其中包含了纽约城（New York City）中每家餐厅的菜单数据和坐标，有了餐厅位置的索引，你就可以查询数据库，找到离布鲁克林大桥（Brooklyn Bridge）最近的提供鱼子酱的餐厅。

而且，空间索引器足够通用，足以适用于地球坐标以外的场景。也就是说，你甚至可以用它来索引二维坐标平面或者是火星上的位置。[②]无论什么样的场景，空间索引都相对容易构建与查询。此处我会描述如何构建空间索引、可执行的查询的范围，以及一些内部设计细节。

E.1　空间索引基础知识

我们将使用美国邮政编码数据库来演示MongoDB的空间索引，你可以从http://mng.bz/dOpd获得该数据。对压缩包解压之后，就能得到一个JSON文件，可以通过`mongoimport`将其导入MongoDB，就像这样：

```
$ mongoimport -d geo -c zips zips.json
```

让我们先看一个邮编文档，如果按照导入指南操作，应该能像下面这样获取文档：

```
> use geo
> db.zips.findOne({zip: 10011})
{
"_id" : ObjectId("4d291187888cec7267e55d24"),
"city" : "New York City",
```

① 其中最著名的就是Foursquare（http://foursquare.com）。从http://mng.bz/rh4n可以了解到更多Foursquare使用MongoDB的情况。

② 前者有一个很好的例子——WordSquared（http://wordsquared.com），这是一个类似Scrabble的游戏，使用了MongoDB的空间索引对棋盘上的格子进行查询。

```
"loc" : {
  "lon" : -73.9996
  "lat" : 40.7402,
},
"state" : "New York",
"zip" : 10011
}
```

除了期望中的城市、州和邮编字段，还有第四个字段loc，其中保存了指定邮编地区的地理中心坐标，这就是你想索引并查询的字段。只有那些包含了坐标值的字段能进行空间索引，但请注意，字段的形式并不算太严格。你可以使用不同的键来表示这些坐标：

```
{ "loc" : { "x" : -73.9996, "y" : 40.7402 } }
```

或者用简单的数组对：

```
{ "loc" : [ -73.9996, 40.7402  ] }
```

只要使用了其中包含两个值的子文档或者数组，这样的字段就能进行空间索引。

现在来创建索引，将索引类型指定为2d：

```
> use geo
> db.zips.ensureIndex({loc: '2d'})
```

这会在loc字段上构建出一个空间索引。只有那些包含恰当格式坐标对的文档会被索引，因此空间索引总是稀疏的。默认的最小和最大坐标值分别是-180和180。这是地理坐标的范围，但如果正好在索引一个不同的领域，可以像下面这样设置最小值和最大值：

```
> use games
> db.moves.ensureIndex({loc: '2d'}, {min: 0, max: 64})
```

一旦构建好了空间索引，就能执行空间查询了。[①]最简单且最常用的空间查询类型是$near查询。当与limit连用时，$near查询允许查找离指定坐标第*n*近的位置。例如，要找到三个离大中央车站（Grand Central Station）最近的邮编，可以发起如下查询：

```
> db.zips.find({'loc': {$near: [ -73.977842, 40.752315 ]}}).limit(3)
{ "_id" : ObjectId("4d291187888cec7267e55d8d"), "city" : "New York City",
  "loc" : { "lon" : -73.9768, "lat" : 40.7519 },
  "state" : "New York", "zip" : 10168 }
{ "_id" : ObjectId("4d291187888cec7267e55d97"), "city" : "New York City",
  "loc" : { "lon" : -73.9785, "lat" : 40.7514 },
  "state" : "New York", "zip" : 10178 }
{ "_id" : ObjectId("4d291187888cec7267e55d8a"), "city" : "New York City",
  "loc" : { "lon" : -73.9791, "lat" : 40.7524 },
  "state" : "New York", "zip" : 10165 }
```

指定一个合理的limit值能确保最快的查询响应时间。如果不做限制，那么会自动将limit值设置为100，这样可以避免返回整个数据集。如果要求返回超过100个结果，可以为limit指定一个数字：

```
> db.zips.find({'loc': {$near: [ -73.977842, 40.752315 ]}}).limit(500)
```

① 注意，空间查询与非空间查询截然不同，它能在查询时指定用或者不用索引。

E.2　高级查询

虽然$near查询适用于很多场景，但MongoDB还提供了一些更高级的查询技术。你可以运行名为geoNear的特殊命令来取代查询，它会返回附近每个对象的距离以及查询本身的一些统计信息：

```
> db.runCommand({'geoNear': 'zips', near: [-73.977842, 40.752315], num: 2})
{
  "ns" : "geo.zips",
  "near" : "0110000111011010011111010110010011001111111011011100",
  "results" : [
    {
      "dis" : 0.00112166376445928 7,
      "obj" : {
        "_id" : ObjectId("4d291187888cec7267e55d8d"),
        "city" : "New York City",
        "loc" : {
          "lon" : -73.9768,
          "lat" : 40.7519
        },
        "state" : "New York",
        "zip" : 10168
      }
    },
    {
      "dis" : 0.001126847051610947,
      "obj" : {
        "_id" : ObjectId("4d291187888cec7267e55d97"),
        "city" : "New York City",
        "loc" : {
          "lon" : -73.9785,
          "lat" : 40.7514
        },
        "state" : "New York",
        "zip" : 10178
      }
    }
  ],
  "stats" : {
    "time" : 0,
    "btreelocs" : 4,
    "nscanned" : 3,
    "objectsLoaded" : 2,
    "avgDistance" : 0.001124255408035117,
    "maxDistance" : 0.001126847051610947
  },
  "ok" : 1
}
```

每个文档中的dis字段用来测量它与中心点的距离，这里的距离是用度数来测量的。

另一个稍微高级点的查询，允许通过$within查询操作符来搜索特定边界内的结果。例如，要查找所有离大中央车站在0.011度范围内的邮编，可以发起如下$center查询：

```
> center = [-73.977842, 40.752315]
> radius = 0.011
> db.zips.find({loc: {$within: {$center: [ center, radius ] }}}).count()
26
```

这在理论上等价于运行带有可选的`$maxDistance`参数的`$near`查询。这两条查询都能返回离中心点指定距离内的所有结果。

```
> db.zips.find({'loc': {$near: [-73.977842, 40.752315],
  $maxDistance: 0.011}}).count()
26
```

除了`$center`操作，还可以使用`$box`操作符来返回特定包围盒（bounding box）内的结果。例如，要返回大中央车站和拉瓜地亚机场（LaGuardia Airport）包围盒内的所有邮编，可以发起如下查询：

```
> lower_left = [-73.977842, 40.752315]
> upper_right = [-73.923649, 40.762925]
> db.zips.find({loc: {$within:
  {$box: [ lower_left, upper_right ] }}}).count()
15
```

请注意，`$box`操作符要求有一个双元素数组，第一个元素是包围盒的左下角坐标，第二个元素是右上角坐标。

E.3　复合空间索引

可以创建复合空间索引，只要坐标键放在第一位即可。你可以使用复合空间索引来对位置和其他一些类型的元数据进行查询。举例来说，假设本书之前介绍的园艺店有不同的零售店位置，不同的店提供不同的服务。两个位置文档片段可能是这样的：

```
{loc: [-74.2, 40.3], services: ['nursery', 'rentals']}
{loc: [-75.2, 39.3], services: ['rentals']}
```

为了对两个位置和服务进行高效查询，可以创建如下复合索引：

```
> db.locations.ensureIndex({loc: '2d', services: 1})
```

这让对所有销售苗圃的零售店的查找稍显烦琐了：

```
> db.locations.find({loc: [-73.977842, 40.752315], services: 'nursery'})
```

关于复合空间索引，除此之外没有其他的了。如果你还心存疑虑，不知复合空间索引对你的应用程序而言是否足够高效，请试着在相关查询上运行`explain()`。

E.4　球面几何学

目前为止，我所描述的所有空间查询中都使用了扁平的地球模型来进行距离计算。尤其是数据库使用了欧几里德距离（Euclidean distance）[1]来确定两点间的距离。对于很多场景而言，包括

① 参见http://en.wikipedia.org/wiki/Euclidean_distance。

查找距指定点最近的n个位置，这是完全可接受的，数学的简单性保证了最快的查询结果。

但现实中，地球大致上是球形的。[1]这意味着欧几里德距离计算变得越来越不精确，因为两点间的距离变大了。为此，MongoDB也支持基于二维球状模型进行距离计算。这些查询会得到更准确的距离，只是会有稍许性能开销。

要使用球面几何学，你只需保证所有坐标都是按照经度-纬度顺序排列，所有距离都是用弧度来表示的。之前的大多数查询都可以表示为球面形式。例如，`$nearSphere`就是`$near`的球面等价形式，表示如下：

```
> db.zips.find({'loc': {$nearSphere: [ -73.977842, 40.752315 ]}}).limit(3)
```

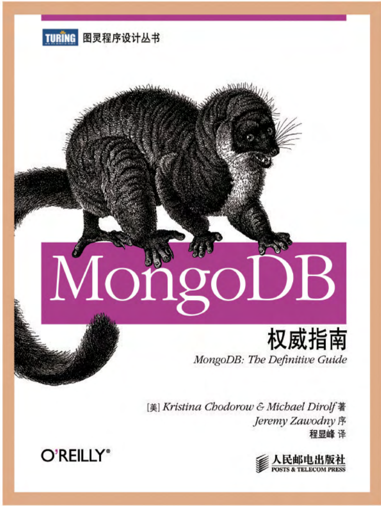
`geoNear`命令还支持球面计算，带上`{ spherical: true }`选项：

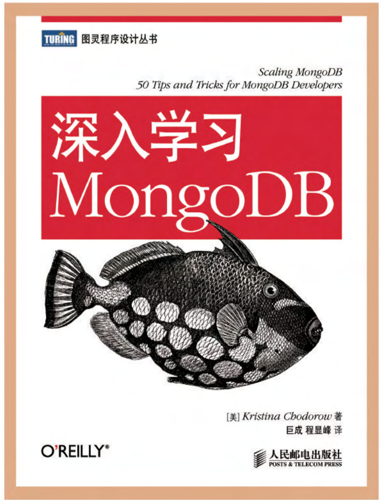
```
> db.runCommand({'geoNear': 'zips',
  near: [-73.977842, 40.752315], num: 2, spherical: true})
```

最后，可以使用球面距离，通过`$centerSphere`对一个圈进行查询。只要保证在指定半径时使用弧度：

```
center = [-73.977842, 40.752315]
radius_in_degrees = 0.11
radius_in_radians = radius_in_degrees * (Math.PI / 180);
db.zips.find({loc: {$within:
  {$centerSphere: [center, radius_in_radians ] }}})
```

① 大体上来说，地球严格上是扁球面的，在赤道上有点凸起。


TURING 图灵程序设计丛书
MongoDB
权威指南
MongoDB: The Definitive Guide
[美] Kristina Chodorow & Michael Dirolf 著
Jeremy Zawodny 序
程显峰 译
O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS


TURING 图灵程序设计丛书
Scaling MongoDB
50 Tips and Tricks for MongoDB Developers
深入学习
MongoDB
[美] Kristina Chodorow 著
巨成 程显峰 译
O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

MongoDB in Action

MongoDB实战



“作者是10gen的人，对所有细节都了如指掌。读这本书，就好像跟一位领域专家对话，一切都讲得那么简洁明了，浅显易懂。所有MongoDB用户都应该看一看。”

“与市面上其他同类主题的书相比，这本书是最好的。”

——亚马逊读者评论

MongoDB是为处理大数据而生的一款面向文档的数据库，由10gen公司开发和维护。本书作者Kyle Banker曾在该公司负责MongoDB驱动程序的维护，对各方面技术细节都了如指掌，本书也是在大量第一手资料的基础上形成的，其权威性毋庸置疑。

本书基于MongoDB 2.0+，全面系统地讲解了设计、实现、安装和维护MongoDB的各方面内容。全书分三部分，第一部分从基于文档的数据与传统关系型数据库的差别讲起，介绍了MongoDB的基本概念及安装使用。第二部分是一个实战式教程，结合示例讲解了MongoDB的CRUD操作，以及实现系统安全、灵活和高效的设计原则及模式。第三部分侧重数据库的维护和管理，深入到MongoDB背后的技术细节，给出了对管理员和开发者都极有价值的建议。

本书篇幅适中，内容深浅得当，文字通俗易懂，再配以直观形象的插图和贴近实战的代码示例，非常适合MongoDB学习者、开发人员及管理员学习参考。

本书内容

- MongoDB介绍及其优劣势
- MongoDB的Shell界面
- 使用MongoDB的简单应用
- 如何通过以文档为中心的方式看待数据
- 编写查询，以MapReduce方式聚合数据
- 更新和删除数据及相关性能考量
- 寻找和改进慢查询
- MongoDB的复制与分片
- MongoDB的监控、备份及恢复



图灵社区：www.ituring.com.cn

新浪微博：@图灵教育　@图灵社区

反馈/投稿/推荐信箱：contact@turingbook.com

热线：(010)51095186转604

分类建议　计算机/数据库/MongoDB

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-29507-1

定价：59.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn